Python Documentation contents

- What's New in Python
  - What's New In Python 3.7
    - Summary – Release Highlights
    - New Features
      - PEP 563: Postponed Evaluation of Annotations
      - PEP 538: Legacy C Locale Coercion
      - PEP 540: Forced UTF-8 Runtime Mode
      - PEP 553: Built-in breakpoint()
      - PEP 539: New C API for Thread-Local Storage
      - PEP 562: Customization of Access to Module Attributes
      - PEP 564: New Time Functions With Nanosecond Resolution
      - PEP 565: Show DeprecationWarning in __main__
      - PEP 560: Core Support for typing module and Generic Types
      - PEP 552: Hash-based .pyc Files
      - PEP 545: Python Documentation Translations
      - Development Runtime Mode: -X dev
  - Other Language Changes
  - New Modules
    - contextvars
    - dataclasses
    - importlib.resources
  - Improved Modules
    - argparse
    - asyncio
    - binascii
    - calendar
    - collections
- compileall
- concurrent.futures
- contextlib
- cProfile
- crypt
- datetime
- dbm
- decimal
- dis
- distutils
- enum
- functools
- gc
- hmac
- http.client
- http.server
- idlelib and IDLE
- importlib
- io
- ipaddress
- itertools
- locale
- logging
- math
- mimetypes
- msilib
- multiprocessing
- os
- pathlib
- pdb
- py_compile
- pydoc
- queue
- re
- signal
- socket
- socketserver
- sqlite3
- ssl
- string
- subprocess
- sys
- time
- tkinter
- tracemalloc
- types
- unicodedata
- unittest
- unittest.mock
- urllib.parse
- uu
- uuid
- warnings
- xml
- xml.etree
- xmlrpc.server
- zipapp
- zipfile

- C API Changes
- Build Changes
- Optimizations
- Other CPython Implementation Changes
- Deprecated Python Behavior
- Deprecated Python modules, functions and methods
  - aifc
  - asyncio
  - collections
  - dbm
- enum
- gettext
- importlib
- locale
- macpath
- threading
- socket
- ssl
- sunau
- sys
- wave

- Deprecated functions and types of the C API
- Platform Support Removals
- API and Feature Removals
- Module Removals
- Windows-only Changes
- Porting to Python 3.7
  - Changes in Python Behavior
  - Changes in the Python API
  - Changes in the C API
  - CPython bytecode changes
  - Windows-only Changes
  - Other CPython implementation changes
- Notable changes in Python 3.7.1
- Notable changes in Python 3.7.2

- What's New In Python 3.6
  - Summary – Release highlights
  - New Features
    - PEP 498: Formatted string literals
    - PEP 526: Syntax for variable annotations
    - PEP 515: Underscores in Numeric Literals
    - PEP 525: Asynchronous Generators
    - PEP 530: Asynchronous Comprehensions
    - PEP 487: Simpler customization of class creation
- PEP 487: Descriptor Protocol Enhancements
- PEP 519: Adding a file system path protocol
- PEP 495: Local Time Disambiguation
- PEP 529: Change Windows filesystem encoding to UTF-8
- PEP 528: Change Windows console encoding to UTF-8
- PEP 520: Preserving Class Attribute Definition Order
- PEP 468: Preserving Keyword Argument Order
- New `dict` implementation
- PEP 523: Adding a frame evaluation API to CPython
- PYTHONMALLOC environment variable
- DTrace and SystemTap probing support

- Other Language Changes
- New Modules
  - secrets
- Improved Modules
  - array
  - ast
  - asyncio
  - binascii
  - cmath
  - collections
  - concurrent.futures
  - contextlib
  - datetime
  - decimal
  - distutils
  - email
  - encodings
  - enum
  - faulthandler
  - fileinput
- hashlib
- http.client
- idlelib and IDLE
- importlib
- inspect
- json
- logging
- math
- multiprocessing
- os
- pathlib
- pdb
- pickle
- pickletools
- pydoc
- random
- re
- readline
- rlcompleter
- shlex
- site
- sqlite3
- socket
- socketserver
- ssl
- statistics
- struct
- subprocess
- sys
- telnetlib
- time
- timeit
- tkinter
- traceback
- tracemalloc
- typing
- unicodedata
- unittest.mock
- urllib.request
- urllib.robotparser
- venv
- warnings
- winreg
- winsound
- xmlrpc.client
- zipfile
- zlib

- Optimizations
- Build and C API Changes
- Other Improvements
- Deprecated
  - New Keywords
  - Deprecated Python behavior
  - Deprecated Python modules, functions and methods
    - asynchat
    - asyncore
    - dbm
    - distutils
    - grp
    - importlib
    - os
    - re
    - ssl
    - tkinter
    - venv
  - Deprecated functions and types of the C API
  - Deprecated Build Options
- Removed
- API and Feature Removals
- Porting to Python 3.6
  - Changes in ‘python’ Command Behavior
  - Changes in the Python API
  - Changes in the C API
  - CPython bytecode changes
- Notable changes in Python 3.6.2
  - New make `regen-all` build target
  - Removal of make `touch` build target
- Notable changes in Python 3.6.4
- Notable changes in Python 3.6.5
- Notable changes in Python 3.6.7
  - What’s New In Python 3.5
    - Summary – Release highlights
    - New Features
      - PEP 492 - Coroutines with async and await syntax
      - PEP 465 - A dedicated infix operator for matrix multiplication
      - PEP 448 - Additional Unpacking Generalizations
      - PEP 461 - percent formatting support for bytes and bytearray
      - PEP 484 - Type Hints
      - PEP 471 - os.scandir() function – a better and faster directory iterator
      - PEP 475: Retry system calls failing with EINTR
      - PEP 479: Change StopIteration handling inside generators
      - PEP 485: A function for testing approximate equality
      - PEP 486: Make the Python Launcher aware of virtual environments
      - PEP 488: Elimination of PYO files
      - PEP 489: Multi-phase extension module initialization
- Other Language Changes
- New Modules
* typing
* zipapp

**Improved Modules**
* argparse
* asyncio
* bz2
* cgi
* cmath
* code
* collections
* collections.abc
* compileall
* concurrent.futures
* configparser
* contextlib
* csv
* curses
* dbm
* difflib
* distutils
* doctest
* email
* enum
* faulthandler
* functools
* glob
* gzip
* heapq
* http
* http.client
* idlelib and IDLE
* imaplib
* imghdr
* importlib
- inspect
- io
- ipaddress
- json
- linecache
- locale
- logging
- lzma
- math
- multiprocessing
- operator
- os
- pathlib
- pickle
- poplib
- re
- readline
- selectors
- shutil
- signal
- smtpd
- smtplib
- sndhdr
- socket
- ssl
  - Memory BIO Support
  - Application-Layer Protocol Negotiation Support
  - Other Changes
- sqlite3
- subprocess
- sys
- sysconfig
- tarfile
- threading
- time
- timeit
- tkinter
- traceback
- types
- unicodedata
- unittest
- unittest.mock
- urllib
- wsgiref
- xmlrpc
- xml.sax
- zipfile

- Other module-level changes
- Optimizations
- Build and C API Changes
- Deprecated
  - New Keywords
  - Deprecated Python Behavior
  - Unsupported Operating Systems
  - Deprecated Python modules, functions and methods
- Removed
  - API and Feature Removals
- Porting to Python 3.5
  - Changes in Python behavior
  - Changes in the Python API
  - Changes in the C API
- Notable changes in Python 3.5.4
  - New make regen-all build target
  - Removal of make touch build target

- What's New In Python 3.4
  - Summary – Release Highlights
  - New Features
    - PEP 453: Explicit Bootstrapping of PIP in Python
Installations
  ■ Bootstrapping pip By Default
  ■ Documentation Changes
    ■ PEP 446: Newly Created File Descriptors Are Non-Inheritable
    ■ Improvements to Codec Handling
    ■ PEP 451: A ModuleSpec Type for the Import System
    ■ Other Language Changes

■ New Modules
  ■ asyncio
  ■ ensurepip
  ■ enum
  ■ pathlib
  ■ selectors
  ■ statistics
  ■ tracemalloc

■ Improved Modules
  ■ abc
  ■ aifc
  ■ argparse
  ■ audioop
  ■ base64
  ■ collections
  ■ colorsys
  ■ contextlib
  ■ dbm
  ■ dis
  ■ doctest
  ■ email
  ■ filecmp
  ■ functools
  ■ gc
  ■ glob
  ■ hashlib
hmac
html
http
idlelib and IDLE
importlib
inspect
ipaddress
logging
marshal
mmap
multiprocessing
operator
os
pdb
pickle
plistlib
poplib
pprint
pty
pydoc
re
resource
select
shelve
shutil
smtpd
smtplib
socket
sqlite3
ssl
stat
struct
subprocess
sunau
- sys
- tarfile
- textwrap
- threading
- traceback
- types
- urllib
- unittest
- venv
- wave
- weakref
- xml.etree
- zipfile

**CPython Implementation Changes**

- PEP 445: Customization of CPython Memory Allocators
- PEP 442: Safe Object Finalization
- PEP 456: Secure and Interchangeable Hash Algorithm
- PEP 436: Argument Clinic

**Other Build and C API Changes**

**Other Improvements**

**Significant Optimizations**

**Deprecated**

- Deprecations in the Python API
- Deprecated Features

**Removed**

- Operating Systems No Longer Supported
- API and Feature Removals
- Code Cleanups

**Porting to Python 3.4**

- Changes in ‘python’ Command Behavior
- Changes in the Python API
- Changes in the C API
- Changed in 3.4.3
  - PEP 476: Enabling certificate verification by default for stdlib http clients
- What's New In Python 3.3
  - Summary – Release highlights
  - PEP 405: Virtual Environments
  - PEP 420: Implicit Namespace Packages
  - PEP 3118: New memoryview implementation and buffer protocol documentation
    - Features
    - API changes
  - PEP 393: Flexible String Representation
    - Functionality
    - Performance and resource usage
  - PEP 397: Python Launcher for Windows
  - PEP 3151: Reworking the OS and IO exception hierarchy
  - PEP 380: Syntax for Delegating to a Subgenerator
  - PEP 409: Suppressing exception context
  - PEP 414: Explicit Unicode literals
  - PEP 3155: Qualified name for classes and functions
  - PEP 412: Key-Sharing Dictionary
  - PEP 362: Function Signature Object
  - PEP 421: Adding sys.implementation
    - SimpleNamespace
  - Using importlib as the Implementation of Import
    - New APIs
    - Visible Changes
  - Other Language Changes
  - A Finer-Grained Import Lock
  - Built-in functions and types
  - New Modules
    - faulthandler
    - ipaddress
- lzma
- **Improved Modules**
  - abc
  - array
  - base64
  - binascii
  - bz2
  - codecs
  - collections
  - contextlib
  - crypt
  - curses
  - datetime
  - decimal
    - Features
    - API changes
- email
  - Policy Framework
  - Provisional Policy with New Header API
  - Other API Changes
- ftplib
- functools
- gc
- hmac
- http
- html
- imaplib
- inspect
- io
- itertools
- logging
- math
- mmap
- multiprocessing
- nntplib
- os
- pdb
- pickle
- pydoc
- re
- sched
- select
- shlex
- shutil
- signal
- smtpd
- smtplib
- socket
- socketserver
- sqlite3
- ssl
- stat
- struct
- subprocess
- sys
- tarfile
- tempfile
- textwrap
- threading
- time
- types
- unittest
- urllib
- webbrowser
- xml.etree.ElementTree
- zlib

- Optimizations
- Build and C API Changes
- Deprecated
  - Unsupported Operating Systems
  - Deprecated Python modules, functions and methods
  - Deprecated functions and types of the C API
  - Deprecated features
- Porting to Python 3.3
  - Porting Python code
  - Porting C code
  - Building C extensions
  - Command Line Switch Changes
- What's New In Python 3.2
  - PEP 384: Defining a Stable ABI
  - PEP 389: Argparse Command Line Parsing Module
  - PEP 391: Dictionary Based Configuration for Logging
  - PEP 3148: The concurrent.futures module
  - PEP 3147: PYC Repository Directories
  - PEP 3149: ABI Version Tagged .so Files
  - PEP 3333: Python Web Server Gateway Interface v1.0.1
- Other Language Changes
- New, Improved, and Deprecated Modules
  - email
  - elementtree
  - functools
  - itertools
  - collections
  - threading
  - datetime and time
  - math
  - abc
  - io
  - reprlib
  - logging
  - csv
  - contextlib
- decimal and fractions
- ftp
- popen
- select
- gzip and zipfile
- tarfile
- hashlib
- ast
- os
- shutil
- sqlite3
- html
- socket
- ssl
- nntp
- certificates
- imaplib
- http.client
- unittest
- random
- poplib
- asyncore
- tempfile
- inspect
- pydoc
- dis
- dbm
- ctypes
- site
- sysconfig
- pdb
- configparser
- urllib.parse
- mailbox
- turtledemo
- Multi-threading
- Optimizations
- Unicode
- Codecs
- Documentation
- IDLE
- Code Repository
- Build and C API Changes
- Porting to Python 3.2

- What's New In Python 3.1
  - PEP 372: Ordered Dictionaries
  - PEP 378: Format Specifier for Thousands Separator
  - Other Language Changes
  - New, Improved, and Deprecated Modules
  - Optimizations
  - IDLE
  - Build and C API Changes
  - Porting to Python 3.1

- What's New In Python 3.0
  - Common Stumbling Blocks
    - Print Is A Function
    - Views And Iterators Instead Of Lists
    - Ordering Comparisons
    - Integers
    - Text Vs. Data Instead Of Unicode Vs. 8-bit
  - Overview Of Syntax Changes
    - New Syntax
    - Changed Syntax
    - Removed Syntax
  - Changes Already Present In Python 2.6
  - Library Changes
  - PEP 3101: A New Approach To String Formatting
  - Changes To Exceptions
- Miscellaneous Other Changes
  - Operators And Special Methods
  - Builtins
- Build and C API Changes
- Performance
- Porting To Python 3.0
- What's New in Python 2.7
  - The Future for Python 2.x
  - Changes to the Handling of Deprecation Warnings
  - Python 3.1 Features
  - PEP 372: Adding an Ordered Dictionary to collections
  - PEP 378: FormatSpecifier for Thousands Separator
  - PEP 389: The argparse Module for Parsing Command Lines
  - PEP 391: Dictionary-Based Configuration For Logging
  - PEP 3106: Dictionary Views
  - PEP 3137: The memoryview Object
- Other Language Changes
  - Interpreter Changes
  - Optimizations
- New and Improved Modules
  - New module: importlib
  - New module: sysconfig
  - ttk: Themed Widgets for Tk
  - Updated module: unitest
  - Updated module: ElementTree 1.3
- Build and C API Changes
  - Capsules
  - Port-Specific Changes: Windows
  - Port-Specific Changes: Mac OS X
  - Port-Specific Changes: FreeBSD
- Other Changes and Fixes
- Porting to Python 2.7
- New Features Added to Python 2.7 Maintenance
Releases

- Two new environment variables for debug mode
- PEP 434: IDLE Enhancement Exception for All Branches
- PEP 466: Network Security Enhancements for Python 2.7
- PEP 477: Backport ensurepip (PEP 453) to Python 2.7
  - Bootstrapping pip By Default
  - Documentation Changes
- PEP 476: Enabling certificate verification by default for stdlib http clients
- PEP 493: HTTPS verification migration tools for Python 2.7
- New make regen-all build target
- Removal of make touch build target

Acknowledgements

What's New in Python 2.6

- Python 3.0
- Changes to the Development Process
  - New Issue Tracker: Roundup
  - New Documentation Format: reStructuredText Using Sphinx
- PEP 343: The 'with' statement
  - Writing Context Managers
  - The contextlib module
- PEP 366: Explicit Relative Imports From a Main Module
- PEP 370: Per-user site-packages Directory
- PEP 371: The multiprocessing Package
- PEP 3101: Advanced String Formatting
- PEP 3105: print As a Function
- PEP 3110: Exception-Handling Changes
- PEP 3112: Byte Literals
- PEP 3116: New I/O Library
- PEP 3118: Revised Buffer Protocol
- PEP 3119: Abstract Base Classes
- PEP 3127: Integer Literal Support and Syntax
- PEP 3129: Class Decorators
- PEP 3141: A Type Hierarchy for Numbers
  - The fractions Module

Other Language Changes
- Optimizations
- Interpreter Changes

New and Improved Modules
- The ast module
- The future_builtins module
- The json module: JavaScript Object Notation
- The plistlib module: A Property-List Parser
- ctypes Enhancements
- Improved SSL Support

Deprecations and Removals

Build and C API Changes
- Port-Specific Changes: Windows
- Port-Specific Changes: Mac OS X
- Port-Specific Changes: IRIX

Porting to Python 2.6

Acknowledgements

○ What's New in Python 2.5
- PEP 308: Conditional Expressions
- PEP 309: Partial Function Application
- PEP 314: Metadata for Python Software Packages v1.1
- PEP 328: Absolute and Relative Imports
- PEP 338: Executing Modules as Scripts
- PEP 341: Unified try/except/finally
- PEP 342: New Generator Features
- PEP 343: The ‘with’ statement
  - Writing Context Managers
  - The contextlib module
- PEP 352: Exceptions as New-Style Classes
- PEP 353: Using ssize_t as the index type
- PEP 357: The ‘__index__’ method
- Other Language Changes
  - Interactive Interpreter Changes
  - Optimizations
- New, Improved, and Removed Modules
  - The ctypes package
  - The ElementTree package
  - The hashlib package
  - The sqlite3 package
  - The wsgiref package
- Build and C API Changes
  - Port-Specific Changes
- Porting to Python 2.5
- Acknowledgements
  - What's New in Python 2.4
    - PEP 218: Built-In Set Objects
    - PEP 237: Unifying Long Integers and Integers
    - PEP 289: Generator Expressions
    - PEP 292: Simpler String Substitutions
    - PEP 318: Decorators for Functions and Methods
    - PEP 322: Reverse Iteration
    - PEP 324: New subprocess Module
    - PEP 327: Decimal Data Type
      - Why is Decimal needed?
      - The Decimal type
      - The Context type
    - PEP 328: Multi-line Imports
    - PEP 331: Locale-Independent Float/String Conversions
- Other Language Changes
  - Optimizations
- New, Improved, and Deprecated Modules
  - cookielib
• doctest
• Build and C API Changes
  • Port-Specific Changes
• Porting to Python 2.4
• Acknowledgements
  ○ What's New in Python 2.3
    • PEP 218: A Standard Set Datatype
    • PEP 255: Simple Generators
    • PEP 263: Source Code Encodings
    • PEP 273: Importing Modules from ZIP Archives
    • PEP 277: Unicode file name support for Windows NT
    • PEP 278: Universal Newline Support
    • PEP 279: enumerate()
    • PEP 282: The logging Package
    • PEP 285: A Boolean Type
    • PEP 293: Codec Error Handling Callbacks
    • PEP 301: Package Index and Metadata for Distutils
    • PEP 302: New Import Hooks
    • PEP 305: Comma-separated Files
    • PEP 307: Pickle Enhancements
  • Extended Slices
• Other Language Changes
  • String Changes
  • Optimizations
• New, Improved, and Deprecated Modules
  • Date/Time Type
  • The optparse Module
• Pymalloc: A Specialized Object Allocator
• Build and C API Changes
  • Port-Specific Changes
• Other Changes and Fixes
• Porting to Python 2.3
• Acknowledgements
  ○ What's New in Python 2.2
- Introduction
- PEPs 252 and 253: Type and Class Changes
  - Old and New Classes
  - Descriptors
  - Multiple Inheritance: The Diamond Rule
  - Attribute Access
  - Related Links
- PEP 234: Iterators
- PEP 255: Simple Generators
- PEP 237: Unifying Long Integers and Integers
- PEP 238: Changing the Division Operator
- Unicode Changes
- PEP 227: Nested Scopes
- New and Improved Modules
- Interpreter Changes and Fixes
- Other Changes and Fixes
- Acknowledgements

○ What's New in Python 2.1
  - Introduction
  - PEP 227: Nested Scopes
  - PEP 236: __future__ Directives
  - PEP 207: Rich Comparisons
  - PEP 230: Warning Framework
  - PEP 229: New Build System
  - PEP 205: Weak References
  - PEP 232: Function Attributes
  - PEP 235: Importing Modules on Case-Insensitive Platforms
  - PEP 217: Interactive Display Hook
  - PEP 208: New Coercion Model
  - PEP 241: Metadata in Python Packages
  - New and Improved Modules
  - Other Changes and Fixes
  - Acknowledgements
What’s New in Python 2.0
  - Introduction
  - What About Python 1.6?
  - New Development Process
  - Unicode
  - List Comprehensions
  - Augmented Assignment
  - String Methods
  - Garbage Collection of Cycles
  - Other Core Changes
    - Minor Language Changes
    - Changes to Built-in Functions
  - Porting to 2.0
  - Extending/Embedding Changes
  - Distutils: Making Modules Easy to Install
  - XML Modules
    - SAX2 Support
    - DOM Support
    - Relationship to PyXML
  - Module changes
  - New modules
  - IDLE Improvements
  - Deleted and Deprecated Modules
  - Acknowledgements

- Changelog
- The Python Tutorial
  - 1. Whetting Your Appetite
  - 2. Using the Python Interpreter
    - 2.1. Invoking the Interpreter
      - 2.1.1. Argument Passing
      - 2.1.2. Interactive Mode
    - 2.2. The Interpreter and Its Environment
      - 2.2.1. Source Code Encoding
  - 3. An Informal Introduction to Python
3.1. Using Python as a Calculator
   - 3.1.1. Numbers
   - 3.1.2. Strings
   - 3.1.3. Lists
3.2. First Steps Towards Programming
4. More Control Flow Tools
   - 4.1. if Statements
   - 4.2. for Statements
   - 4.3. The range() Function
   - 4.4. break and continue Statements, and else Clauses on Loops
   - 4.5. pass Statements
   - 4.6. Defining Functions
   - 4.7. More on Defining Functions
     - 4.7.1. Default Argument Values
     - 4.7.2. Keyword Arguments
     - 4.7.3. Arbitrary Argument Lists
     - 4.7.4. Unpacking Argument Lists
     - 4.7.5. Lambda Expressions
     - 4.7.6. Documentation Strings
     - 4.7.7. Function Annotations
   - 4.8. Intermezzo: Coding Style
5. Data Structures
   - 5.1. More on Lists
     - 5.1.1. Using Lists as Stacks
     - 5.1.2. Using Lists as Queues
     - 5.1.3. List Comprehensions
     - 5.1.4. Nested List Comprehensions
   - 5.2. The del statement
   - 5.3. Tuples and Sequences
   - 5.4. Sets
   - 5.5. Dictionaries
   - 5.6. Looping Techniques
   - 5.7. More on Conditions
5.8. Comparing Sequences and Other Types

6. Modules

6.1. More on Modules
   - 6.1.1. Executing modules as scripts
   - 6.1.2. The Module Search Path
   - 6.1.3. “Compiled” Python files

6.2. Standard Modules

6.3. The dir() Function

6.4. Packages
   - 6.4.1. Importing * From a Package
   - 6.4.2. Intra-package References
   - 6.4.3. Packages in Multiple Directories

7. Input and Output

7.1. Fancier Output Formatting
   - 7.1.1. Formatted String Literals
   - 7.1.2. The String format() Method
   - 7.1.3. Manual String Formatting
   - 7.1.4. Old string formatting

7.2. Reading and Writing Files
   - 7.2.1. Methods of File Objects
   - 7.2.2. Saving structured data with json

8. Errors and Exceptions

8.1. Syntax Errors

8.2. Exceptions

8.3. Handling Exceptions

8.4. Raising Exceptions

8.5. User-defined Exceptions

8.6. Defining Clean-up Actions

8.7. Predefined Clean-up Actions

9. Classes

9.1. A Word About Names and Objects

9.2. Python Scopes and Namespaces
   - 9.2.1. Scopes and Namespaces Example

9.3. A First Look at Classes
- 9.3.1. Class Definition Syntax
- 9.3.2. Class Objects
- 9.3.3. Instance Objects
- 9.3.4. Method Objects
- 9.3.5. Class and Instance Variables
- 9.4. Random Remarks
- 9.5. Inheritance
  - 9.5.1. Multiple Inheritance
- 9.6. Private Variables
- 9.7. Odds and Ends
- 9.8. Iterators
- 9.9. Generators
- 9.10. Generator Expressions
- 10. Brief Tour of the Standard Library
  - 10.1. Operating System Interface
  - 10.2. File Wildcards
  - 10.3. Command Line Arguments
  - 10.4. Error Output Redirection and Program Termination
  - 10.5. String Pattern Matching
  - 10.6. Mathematics
  - 10.7. Internet Access
  - 10.8. Dates and Times
  - 10.9. Data Compression
  - 10.10. Performance Measurement
  - 10.11. Quality Control
  - 10.12. Batteries Included
- 11. Brief Tour of the Standard Library — Part II
  - 11.1. Output Formatting
  - 11.2. Templating
  - 11.3. Working with Binary Data Record Layouts
  - 11.4. Multi-threading
  - 11.5. Logging
  - 11.6. Weak References
  - 11.7. Tools for Working with Lists
- 11.8. Decimal Floating Point Arithmetic
- 12. Virtual Environments and Packages
  - 12.1. Introduction
  - 12.2. Creating Virtual Environments
  - 12.3. Managing Packages with pip
- 13. What Now?
- 14. Interactive Input Editing and History Substitution
  - 14.1. TabCompletion and History Editing
  - 14.2. Alternatives to the Interactive Interpreter
- 15. Floating Point Arithmetic: Issues and Limitations
  - 15.1. Representation Error
- 16. Appendix
  - 16.1. Interactive Mode
    - 16.1.1. Error Handling
    - 16.1.2. Executable Python Scripts
    - 16.1.3. The Interactive Startup File
    - 16.1.4. The Customization Modules
- Python Setup and Usage
  - 1. Command line and environment
    - 1.1. Command line
      - 1.1.1. Interface options
      - 1.1.2. Generic options
      - 1.1.3. Miscellaneous options
      - 1.1.4. Options you shouldn’t use
    - 1.2. Environment variables
      - 1.2.1. Debug-mode variables
  - 2. Using Python on Unix platforms
    - 2.1. Getting and installing the latest version of Python
      - 2.1.1. On Linux
      - 2.1.2. On FreeBSD and OpenBSD
      - 2.1.3. On OpenSolaris
    - 2.2. Building Python
    - 2.3. Python-related paths and files
    - 2.4. Miscellaneous
2.5. Editors and IDEs

3. Using Python on Windows

3.1. The full installer
- 3.1.1. Installation steps
- 3.1.2. Removing the MAX_PATH Limitation
- 3.1.3. Installing Without UI
- 3.1.4. Installing Without Downloading
- 3.1.5. Modifying an install

3.2. The Microsoft Store package
- 3.2.1. Known Issues

3.3. The nuget.org packages

3.4. The embeddable package
- 3.4.1. Python Application
- 3.4.2. Embedding Python

3.5. Alternative bundles

3.6. Configuring Python
- 3.6.1. Excursus: Setting environment variables
- 3.6.2. Finding the Python executable

3.7. Python Launcher for Windows
- 3.7.1. Getting started
  - 3.7.1.1. From the command-line
  - 3.7.1.2. Virtual environments
  - 3.7.1.3. From a script
  - 3.7.1.4. From file associations
- 3.7.2. Shebang Lines
- 3.7.3. Arguments in shebang lines
- 3.7.4. Customization
  - 3.7.4.1. Customization via INI files
  - 3.7.4.2. Customizing default Python versions
- 3.7.5. Diagnostics

3.8. Finding modules

3.9. Additional modules
- 3.9.1. PyWin32
- 3.9.2. cx_Freeze
3.9.3. WConio
3.10. Compiling Python on Windows
3.11. Other Platforms
4. Using Python on a Macintosh
  4.1. Getting and Installing MacPython
    4.1.1. How to run a Python script
    4.1.2. Running scripts with a GUI
    4.1.3. Configuration
  4.2. The IDE
  4.3. Installing Additional Python Packages
  4.4. GUI Programming on the Mac
  4.5. Distributing Python Applications on the Mac
  4.6. Other Resources

The Python Language Reference
  1. Introduction
    1.1. Alternate Implementations
    1.2. Notation
  2. Lexical analysis
    2.1. Line structure
      2.1.1. Logical lines
      2.1.2. Physical lines
      2.1.3. Comments
      2.1.4. Encoding declarations
      2.1.5. Explicit line joining
      2.1.6. Implicit line joining
      2.1.7. Blank lines
      2.1.8. Indentation
      2.1.9. Whitespace between tokens
    2.2. Other tokens
    2.3. Identifiers and keywords
      2.3.1. Keywords
      2.3.2. Reserved classes of identifiers
    2.4. Literals
      2.4.1. String and Bytes literals
- 2.4.2. String literal concatenation
- 2.4.3. Formatted string literals
- 2.4.4. Numeric literals
- 2.4.5. Integer literals
- 2.4.6. Floating point literals
- 2.4.7. Imaginary literals
- 2.5. Operators
- 2.6. Delimiters

3. Data model
- 3.1. Objects, values and types
- 3.2. The standard type hierarchy
- 3.3. Special method names
  - 3.3.1. Basic customization
  - 3.3.2. Customizing attribute access
    - 3.3.2.1. Customizing module attribute access
    - 3.3.2.2. Implementing Descriptors
    - 3.3.2.3. Invoking Descriptors
    - 3.3.2.4. __slots__
      - 3.3.2.4.1. Notes on using __slots__
  - 3.3.3. Customizing class creation
    - 3.3.3.1. Metaclasses
    - 3.3.3.2. Resolving MRO entries
    - 3.3.3.3. Determining the appropriate metaclass
    - 3.3.3.4. Preparing the class namespace
    - 3.3.3.5. Executing the class body
    - 3.3.3.6. Creating the class object
    - 3.3.3.7. Uses for metaclasses
  - 3.3.4. Customizing instance and subclass checks
  - 3.3.5. Emulating generic types
  - 3.3.6. Emulating callable objects
  - 3.3.7. Emulating container types
  - 3.3.8. Emulating numeric types
  - 3.3.9. With Statement Context Managers
  - 3.3.10. Special method lookup
3.4. Coroutines
   - 3.4.1. Awaitable Objects
   - 3.4.2. Coroutine Objects
   - 3.4.3. Asynchronous Iterators
   - 3.4.4. Asynchronous Context Managers

4. Execution model
   - 4.1. Structure of a program
   - 4.2. Naming and binding
     - 4.2.1. Binding of names
     - 4.2.2. Resolution of names
     - 4.2.3. Builtins and restricted execution
     - 4.2.4. Interaction with dynamic features
   - 4.3. Exceptions

5. The import system
   - 5.1. importlib
   - 5.2. Packages
     - 5.2.1. Regular packages
     - 5.2.2. Namespace packages
   - 5.3. Searching
     - 5.3.1. The module cache
     - 5.3.2. Finders and loaders
     - 5.3.3. Import hooks
     - 5.3.4. The meta path
   - 5.4. Loading
     - 5.4.1. Loaders
     - 5.4.2. Submodules
     - 5.4.3. Module spec
     - 5.4.4. Import-related module attributes
     - 5.4.5. module.__path__
     - 5.4.6. Module reprs
     - 5.4.7. Cached bytecode invalidation
   - 5.5. The Path Based Finder
     - 5.5.1. Path entry finders
     - 5.5.2. Path entry finder protocol
- 5.6. Replacing the standard import system
- 5.7. Special considerations for __main__
  - 5.7.1. __main__.__spec__
- 5.8. Open issues
- 5.9. References
- 6. Expressions
  - 6.1. Arithmetic conversions
  - 6.2. Atoms
    - 6.2.1. Identifiers (Names)
    - 6.2.2. Literals
    - 6.2.3. Parenthesized forms
    - 6.2.4. Displays for lists, sets and dictionaries
    - 6.2.5. List displays
    - 6.2.6. Set displays
    - 6.2.7. Dictionary displays
    - 6.2.8. Generator expressions
    - 6.2.9. Yield expressions
      - 6.2.9.1. Generator-iterator methods
      - 6.2.9.2. Examples
      - 6.2.9.3. Asynchronous generator functions
      - 6.2.9.4. Asynchronous generator-iterator methods
  - 6.3. Primaries
    - 6.3.1. Attribute references
    - 6.3.2. Subscriptions
    - 6.3.3. Slicings
    - 6.3.4. Calls
  - 6.4. Await expression
  - 6.5. The power operator
  - 6.6. Unary arithmetic and bitwise operations
  - 6.7. Binary arithmetic operations
  - 6.8. Shifting operations
  - 6.9. Binary bitwise operations
  - 6.10. Comparisons
- 6.10.1. Value comparisons
- 6.10.2. Membership test operations
- 6.10.3. Identity comparisons
- 6.11. Boolean operations
- 6.12. Conditional expressions
- 6.13. Lambdas
- 6.14. Expression lists
- 6.15. Evaluation order
- 6.16. Operator precedence

- 7. Simple statements
  - 7.1. Expression statements
  - 7.2. Assignment statements
    - 7.2.1. Augmented assignment statements
    - 7.2.2. Annotated assignment statements
  - 7.3. The assert statement
  - 7.4. The pass statement
  - 7.5. The del statement
  - 7.6. The return statement
  - 7.7. The yield statement
  - 7.8. The raise statement
  - 7.9. The break statement
  - 7.10. The continue statement
  - 7.11. The import statement
    - 7.11.1. Future statements
  - 7.12. The global statement
  - 7.13. The nonlocal statement

- 8. Compound statements
  - 8.1. The if statement
  - 8.2. The while statement
  - 8.3. The for statement
  - 8.4. The try statement
  - 8.5. The with statement
  - 8.6. Function definitions
8.7. Class definitions
8.8. Coroutines
  8.8.1. Coroutine function definition
  8.8.2. The async for statement
  8.8.3. The async with statement
9. Top-level components
  9.1. Complete Python programs
  9.2. File input
  9.3. Interactive input
  9.4. Expression input
10. Full Grammar specification

- The Python Standard Library
  - Introduction
    - Notes on availability
  - Built-in Functions
  - Built-in Constants
    - Constants added by the site module
  - Built-in Types
    - Truth Value Testing
    - Boolean Operations — and, or, not
    - Comparisons
    - Numeric Types — int, float, complex
      - Bitwise Operations on Integer Types
      - Additional Methods on Integer Types
      - Additional Methods on Float
      - Hashing of numeric types
  - Iterator Types
    - Generator Types
  - Sequence Types — list, tuple, range
    - Common Sequence Operations
    - Immutable Sequence Types
    - Mutable Sequence Types
    - Lists
    - Tuples
- Ranges
- **Text Sequence Type — str**
  - String Methods
  - *printf*-style String Formatting
- **Binary Sequence Types — bytes, bytearray, memoryview**
  - Bytes Objects
  - Bytearray Objects
  - Bytes and Bytearray Operations
  - *printf*-style Bytes Formatting
  - Memory Views
- **Set Types — set, frozenset**
- **Mapping Types — dict**
  - Dictionary view objects
- **Context Manager Types**
- **Other Built-in Types**
  - Modules
  - Classes and Class Instances
  - Functions
  - Methods
  - Code Objects
  - Type Objects
  - The Null Object
  - The Ellipsis Object
  - The NotImplemeted Object
  - Boolean Values
  - Internal Objects
- **Special Attributes**
  - **Built-in Exceptions**
    - Base classes
    - Concrete exceptions
      - OS exceptions
    - Warnings
    - Exception hierarchy
Text Processing Services
- **string** — Common string operations
  - String constants
  - Custom String Formatting
  - Format String Syntax
    - Format Specification Mini-Language
    - Format examples
  - Template strings
  - Helper functions
- **re** — Regular expression operations
  - Regular Expression Syntax
  - Module Contents
  - Regular Expression Objects
  - Match Objects
  - Regular Expression Examples
    - Checking for a Pair
    - Simulating scanf()
    - search() vs. match()
    - Making a Phonebook
    - Text Munging
    - Finding all Adverbs
    - Finding all Adverbs and their Positions
    - Raw String Notation
    - Writing a Tokenizer
- **difflib** — Helpers for computing deltas
  - SequenceMatcher Objects
  - SequenceMatcher Examples
  - Differ Objects
  - Differ Example
  - A command-line interface to difflib
- **textwrap** — Text wrapping and filling
- **unicodedata** — Unicode Database
- **stringprep** — Internet String Preparation
- **readline** — GNU readline interface
- Init file
- Line buffer
- History file
- History list
- Startup hooks
- Completion
- Example
  - rlcompleter — Completion function for GNU readline
  - Completer Objects
- Binary Data Services
  - struct — Interpret bytes as packed binary data
    - Functions and Exceptions
    - Format Strings
      - Byte Order, Size, and Alignment
      - Format Characters
      - Examples
  - Classes
- codecs — Codec registry and base classes
- Codec Base Classes
  - Error Handlers
  - Stateless Encoding and Decoding
  - Incremental Encoding and Decoding
    - IncrementalEncoder Objects
    - IncrementalDecoder Objects
  - Stream Encoding and Decoding
    - StreamWriter Objects
    - StreamReader Objects
    - StreamReaderWriter Objects
    - StreamRecoder Objects
- Encodings and Unicode
- Standard Encodings
- Python Specific Encodings
  - Text Encodings
  - Binary Transforms
- Text Transforms
  - `encodings.idna` — Internationalized Domain Names in Applications
  - `encodings.mbc` — Windows ANSI codepage
  - `encodings.utf_8_sig` — UTF-8 codec with BOM signature

- Data Types
  - `datetime` — Basic date and time types
    - Available Types
    - `timedelta` Objects
    - `date` Objects
    - `datetime` Objects
    - `time` Objects
    - `tzinfo` Objects
    - `timezone` Objects
    - `strftime()` and `strptime()` Behavior
  - `calendar` — General calendar-related functions
  - `collections` — Container datatypes
    - `ChainMap` objects
      - `ChainMap` Examples and Recipes
    - `Counter` objects
    - `deque` objects
      - `deque` Recipes
    - `defaultdict` objects
      - `defaultdict` Examples
    - `namedtuple()` Factory Function for Tuples with Named Fields
    - `OrderedDict` objects
      - `OrderedDict` Examples and Recipes
    - `UserDict` objects
    - `UserList` objects
    - `UserString` objects
- `collections.abc` — Abstract Base Classes for Containers
  - Collections Abstract Base Classes
- `heapq` — Heap queue algorithm
  - Basic Examples
  - Priority Queue Implementation Notes
  - Theory
- `bisect` — Array bisection algorithm
  - Searching Sorted Lists
  - Other Examples
- `array` — Efficient arrays of numeric values
- `weakref` — Weak references
  - Weak Reference Objects
  - Example
  - Finalizer Objects
  - Comparing finalizers with `__del__()` methods
- `types` — Dynamic type creation and names for built-in types
  - Dynamic Type Creation
  - Standard Interpreter Types
  - Additional Utility Classes and Functions
  - Coroutine Utility Functions
- `copy` — Shallow and deep copy operations
- `pprint` — Data pretty printer
  - PrettyPrinter Objects
  - Example
- `reprlib` — Alternate `repr()` implementation
  - Repr Objects
  - Subclassing Repr Objects
- `enum` — Support for enumerations
  - Module Contents
  - Creating an Enum
  - Programmatic access to enumeration members and
their attributes

- Duplicating enum members and values
- Ensuring unique enumeration values
- Using automatic values
- Iteration
- Comparisons
- Allowed members and attributes of enumerations
- Restricted Enum subclassing
- Pickling
- Functional API
- Derived Enumerations
  - IntEnum
  - IntFlag
  - Flag
  - Others
- Interesting examples
  - Omitting values
    - Using `auto`
    - Using `object`
    - Using a descriptive string
    - Using a custom `__new__()`
  - OrderedEnum
  - DuplicateFreeEnum
  - Planet
  - TimePeriod
- How are Enums different?
  - Enum Classes
  - Enum Members (aka instances)
  - Finer Points
    - Supported `__dunder__` names
    - Supported `_sunder_` names
    - Enum member type
    - Boolean value of Enum classes and members
- Enum classes with methods
- Combining members of Flag

- Numeric and Mathematical Modules
  - numbers — Numeric abstract base classes
    - The numeric tower
    - Notes for type implementors
      - Adding More Numeric ABCs
      - Implementing the arithmetic operations
  - math — Mathematical functions
    - Number-theoretic and representation functions
    - Power and logarithmic functions
    - Trigonometric functions
    - Angular conversion
    - Hyperbolic functions
    - Special functions
    - Constants
  - cmath — Mathematical functions for complex numbers
    - Conversions to and from polar coordinates
    - Power and logarithmic functions
    - Trigonometric functions
    - Hyperbolic functions
    - Classification functions
    - Constants
  - decimal — Decimal fixed point and floating point arithmetic
    - Quick-start Tutorial
    - Decimal objects
      - Logical operands
    - Context objects
    - Constants
    - Rounding modes
    - Signals
    - Floating Point Notes
      - Mitigating round-off error with increased
- precision
  - Special values
- Working with threads
- Recipes
- Decimal FAQ
- fractions — Rational numbers
- random — Generate pseudo-random numbers
  - Bookkeeping functions
  - Functions for integers
  - Functions for sequences
  - Real-valued distributions
  - Alternative Generator
  - Notes on Reproducibility
  - Examples and Recipes
- statistics — Mathematical statistics functions
  - Averages and measures of central location
  - Measures of spread
  - Function details
  - Exceptions
  - Functional Programming Modules
    - itertools — Functions creating iterators for efficient looping
      - Itertool functions
      - Itertools Recipes
    - functools — Higher-order functions and operations on callable objects
      - partial Objects
    - operator — Standard operators as functions
      - Mapping Operators to Functions
      - Inplace Operators
  - File and Directory Access
    - pathlib — Object-oriented filesystem paths
      - Basic use
      - Pure paths
- General properties
- Operators
- Accessing individual parts
- Methods and properties
  - Concrete paths
    - Methods
  - Correspondence to tools in the os module
- os.path — Common pathname manipulations
- fileinput — Iterate over lines from multiple input streams
- stat — Interpreting stat() results
- filecmp — File and Directory Comparisons
  - The dircmp class
- tempfile — Generate temporary files and directories
  - Examples
  - Deprecated functions and variables
- glob — Unix style pathname pattern expansion
- fnmatch — Unix filename pattern matching
- linecache — Random access to text lines
- shutil — High-level file operations
  - Directory and files operations
    - copytree example
    - rmtree example
  - Archiving operations
    - Archiving example
  - Querying the size of the output terminal
- macpath — Mac OS 9 path manipulation functions

○ Data Persistence
- pickle — Python object serialization
  - Relationship to other Python modules
    - Comparison with marshal
    - Comparison with json
- Data stream format
- Module Interface
- What can be pickled and unpickled?
- Pickling Class Instances
  - Persistence of External Objects
  - Dispatch Tables
  - Handling Stateful Objects
- Restricting Globals
- Performance
- Examples
  - copyreg — Register pickle support functions
    - Example
  - shelve — Python object persistence
    - Restrictions
    - Example
  - marshal — Internal Python object serialization
  - dbm — Interfaces to Unix "databases"
    - dbm.gnu — GNU's reinterpretation of dbm
    - dbm.ndbm — Interface based on ndbm
    - dbm.dumb — Portable DBM implementation
  - sqlite3 — DB-API 2.0 interface for SQLite databases
    - Module functions and constants
    - Connection Objects
    - Cursor Objects
    - Row Objects
    - Exceptions
    - SQLite and Python types
      - Introduction
      - Using adapters to store additional Python types in SQLite databases
        - Letting your object adapt itself
        - Registering an adapter callable
      - Converting SQLite values to custom Python types
      - Default adapters and converters
- Controlling Transactions
- Using sqlite3 efficiently
  - Using shortcut methods
  - Accessing columns by name instead of by index
  - Using the connection as a context manager
- Common issues
  - Multithreading
- Data Compression and Archiving
  - zlib — Compression compatible with gzip
  - gzip — Support for gzip files
    - Examples of usage
  - bz2 — Support for bzip2 compression
    - (De)compression of files
    - Incremental (de)compression
    - One-shot (de)compression
  - lzma — Compression using the LZMA algorithm
    - Reading and writing compressed files
    - Compressing and decompressing data in memory
    - Miscellaneous
    - Specifying custom filter chains
    - Examples
  - zipfile — Work with ZIP archives
    - ZipFile Objects
    - PyZipFile Objects
    - ZipInfo Objects
    - Command-Line Interface
      - Command-line options
  - tarfile — Read and write tar archive files
    - TarFile Objects
    - TarInfo Objects
    - Command-Line Interface
      - Command-line options
    - Examples
    - Supported tar formats
- Unicode issues

- File Formats
  - csv — CSV File Reading and Writing
    - Module Contents
    - Dialects and Formatting Parameters
    - Reader Objects
    - Writer Objects
    - Examples
  - configparser — Configuration file parser
    - Quick Start
    - Supported Datatypes
    - Fallback Values
    - Supported INI File Structure
    - Interpolation of values
    - Mapping Protocol Access
    - Customizing Parser Behaviour
    - Legacy API Examples
    - ConfigParser Objects
    - RawConfigParser Objects
    - Exceptions
  - netrc — netrc file processing
    - netrc Objects
  - xdrlib — Encode and decode XDR data
    - Packer Objects
    - Unpacker Objects
    - Exceptions
  - plistlib — Generate and parse Mac OS X .plist files
    - Examples

- Cryptographic Services
  - hashlib — Secure hashes and message digests
    - Hash algorithms
    - SHAKE variable length digests
    - Key derivation
- BLAKE2
  - Creating hash objects
  - Constants
  - Examples
    - Simple hashing
    - Using different digest sizes
    - Keyed hashing
    - Randomized hashing
    - Personalization
    - Tree mode
- Credits
  - hmac — Keyed-Hashing for Message Authentication
  - secrets — Generate secure random numbers for managing secrets
    - Random numbers
    - Generating tokens
      - How many bytes should tokens use?
    - Other functions
    - Recipes and best practices
- Generic Operating System Services
  - os — Miscellaneous operating system interfaces
    - File Names, Command Line Arguments, and Environment Variables
    - Process Parameters
    - File Object Creation
    - File Descriptor Operations
      - Querying the size of a terminal
      - Inheritance of File Descriptors
    - Files and Directories
      - Linux extended attributes
    - Process Management
    - Interface to the scheduler
    - Miscellaneous System Information
    - Random numbers
- **io** — Core tools for working with streams
  - Overview
    - Text I/O
    - Binary I/O
    - Raw I/O
  - High-level Module Interface
    - In-memory streams
  - Class hierarchy
    - I/O Base Classes
    - Raw File I/O
    - Buffered Streams
    - Text I/O
  - Performance
    - Binary I/O
    - Text I/O
    - Multi-threading
    - Reentrancy
- **time** — Time access and conversions
  - Functions
  - Clock ID Constants
  - Timezone Constants
- **argparse** — Parser for command-line options, arguments and sub-commands
  - Example
    - Creating a parser
    - Adding arguments
    - Parsing arguments
  - ArgumentParser objects
    - prog
    - usage
    - description
    - epilog
    - parents
    - formatter_class
- prefix_chars
- fromfile_prefix_chars
- argument_default
- allow_abbrev
- conflict_handler
- add_help

- The add_argument() method
  - name or flags
  - action
  - nargs
  - const
  - default
  - type
  - choices
  - required
  - help
  - metavar
  - dest
  - Action classes

- The parse_args() method
  - Option value syntax
  - Invalid arguments
  - Arguments containing -
  - Argument abbreviations (prefix matching)
  - Beyond sys.argv
  - The Namespace object

- Other utilities
  - Sub-commands
  - FileType objects
  - Argument groups
  - Mutual exclusion
  - Parser defaults
  - Printing help
  - Partial parsing
- Customizing file parsing
- Exiting methods
- Intermixed parsing
- Upgrading optparse code
- `getopt` — C-style parser for command line options
- `logging` — Logging facility for Python
  - Logger Objects
  - Logging Levels
  - Handler Objects
  - Formatter Objects
  - Filter Objects
  - LogRecord Objects
  - LogRecord attributes
  - LoggerAdapter Objects
  - Thread Safety
  - Module-Level Functions
  - Module-Level Attributes
  - Integration with the warnings module
- `logging.config` — Logging configuration
  - Configuration functions
  - Configuration dictionary schema
    - Dictionary Schema Details
    - Incremental Configuration
    - Object connections
    - User-defined objects
    - Access to external objects
    - Access to internal objects
    - Import resolution and custom importers
  - Configuration file format
- `logging.handlers` — Logging handlers
  - StreamHandler
  - FileHandler
  - NullHandler
  - WatchedFileHandler
- BaseRotatingHandler
- RotatingFileHandler
- TimedRotatingFileHandler
- SocketHandler
- DatagramHandler
- SysLogHandler
- NTEventLogHandler
- SMTPHandler
- MemoryHandler
- HTTPHandler
- QueueHandler
- QueueListener
- getpass — Portable password input
- curses — Terminal handling for character-cell displays
  - Functions
  - Window Objects
  - Constants
- curses.textpad — Text input widget for curses programs
  - Textbox objects
- curses.ascii — Utilities for ASCII characters
- curses.panel — A panel stack extension for curses
  - Functions
  - Panel Objects
- platform — Access to underlying platform's identifying data
  - Cross Platform
  - Java Platform
  - Windows Platform
    - Win95/98 specific
  - Mac OS Platform
  - Unix Platforms
- errno — Standard errno system symbols
- ctypes — A foreign function library for Python
- ctypes tutorial
  - Loading dynamic link libraries
  - Accessing functions from loaded dlls
  - Calling functions
  - Fundamental data types
  - Calling functions, continued
  - Calling functions with your own custom data types
  - Specifying the required argument types (function prototypes)
  - Return types
  - Passing pointers (or: passing parameters by reference)
  - Structures and unions
  - Structure/union alignment and byte order
  - Bit fields in structures and unions
  - Arrays
  - Pointers
  - Type conversions
  - Incomplete Types
  - Callback functions
  - Accessing values exported from dlls
  - Surprises
  - Variable-sized data types

- ctypes reference
  - Finding shared libraries
  - Loading shared libraries
  - Foreign functions
  - Function prototypes
  - Utility functions
  - Data types
  - Fundamental data types
  - Structured data types
  - Arrays and pointers
Concurrent Execution

- threading — Thread-based parallelism
  - Thread-Local Data
  - Thread Objects
  - Lock Objects
  - RLock Objects
  - Condition Objects
  - Semaphore Objects
    - Semaphore Example
  - Event Objects
  - Timer Objects
  - Barrier Objects
  - Using locks, conditions, and semaphores in the with statement

- multiprocessing — Process-based parallelism
  - Introduction
    - The Process class
    - Contexts and start methods
    - Exchanging objects between processes
    - Synchronization between processes
    - Sharing state between processes
    - Using a pool of workers
  - Reference
    - Process and exceptions
    - Pipes and Queues
    - Miscellaneous
    - Connection Objects
    - Synchronization primitives
    - Shared ctypes Objects
      - The multiprocessing.sharedctypes module
    - Managers
      - Customized managers
      - Using a remote manager
- Proxy Objects
  - Cleanup
- Process Pools
- Listeners and Clients
  - Address Formats
- Authentication keys
- Logging
- The multiprocessing.dummy module

- Programming guidelines
  - All start methods
  - The spawn and forkserver start methods

- Examples
- The concurrent package
- concurrent.futures — Launching parallel tasks
  - Executor Objects
  - ThreadPoolExecutor
    - ThreadPoolExecutor Example
  - ProcessPoolExecutor
    - ProcessPoolExecutor Example
- Future Objects
- Module Functions
- Exception classes

- subprocess — Subprocess management
  - Using the subprocess Module
    - Frequently Used Arguments
    - Popen Constructor
    - Exceptions
  - Security Considerations
  - Popen Objects
  - Windows Popen Helpers
    - Windows Constants
  - Older high-level API
  - Replacing Older Functions with the subprocess Module
- Replacing /bin/sh shell backquote
- Replacing shell pipeline
- Replacing os.system()
- Replacing the os.spawn family
- Replacing os.popen(), os.popen2(), os.popen3()
- Replacing functions from the popen2 module
  - Legacy Shell Invocation Functions
  - Notes
    - Converting an argument sequence to a string on Windows
- sched — Event scheduler
  - Scheduler Objects
- queue — A synchronized queue class
  - Queue Objects
  - SimpleQueue Objects
- _thread — Low-level threading API
- _dummy_thread — Drop-in replacement for the _thread module
- dummy_threading — Drop-in replacement for the threading module
- contextvars — Context Variables
  - Context Variables
  - Manual Context Management
  - asyncio support
- Networking and Interprocess Communication
  - asyncio — Asynchronous I/O
    - Coroutines and Tasks
      - Coroutines
      - Awaitables
      - Running an asyncio Program
      - Creating Tasks
      - Sleeping
- Running Tasks Concurrently
- Shielding From Cancellation
- Timeouts
- Waiting Primitives
- Scheduling From Other Threads
- Introspection
- Task Object
- Generator-based Coroutines
- Streams
  - StreamReader
  - StreamWriter
  - Examples
    - TCP echo client using streams
    - TCP echo server using streams
    - Get HTTP headers
    - Register an open socket to wait for data using streams
- Synchronization Primitives
  - Lock
  - Event
  - Condition
  - Semaphore
  - BoundedSemaphore
- Subprocesses
  - Creating Subprocesses
  - Constants
  - Interacting with Subprocesses
    - Subprocess and Threads
    - Examples
- Queues
  - Queue
  - Priority Queue
  - LIFO Queue
  - Exceptions
- Examples
- Exceptions
- Event Loop
  - Event Loop Methods
    - Running and stopping the loop
    - Scheduling callbacks
    - Scheduling delayed callbacks
    - Creating Futures and Tasks
    - Opening network connections
    - Creating network servers
    - Transferring files
    - TLS Upgrade
    - Watching file descriptors
    - Working with socket objects directly
    - DNS
    - Working with pipes
    - Unix signals
    - Executing code in thread or process pools
    - Error Handling API
    - Enabling debug mode
    - Running Subprocesses
- Callback Handles
- Server Objects
- Event Loop Implementations
- Examples
  - Hello World with call_soon()
  - Display the current date with call_later()
  - Watch a file descriptor for read events
  - Set signal handlers for SIGINT and SIGTERM
- Futures
  - Future Functions
  - Future Object
- Transports and Protocols
- Transports
  - Transports Hierarchy
  - Base Transport
  - Read-only Transports
  - Write-only Transports
  - Datagram Transports
  - Subprocess Transports
- Protocols
  - Base Protocols
  - Base Protocol
  - Streaming Protocols
  - Buffered Streaming Protocols
  - Datagram Protocols
  - Subprocess Protocols
- Examples
  - TCP Echo Server
  - TCP Echo Client
  - UDP Echo Server
  - UDP Echo Client
  - Connecting Existing Sockets
  - loop.subprocess_exec() and SubprocessProtocol
- Policies
  - Getting and Setting the Policy
  - Policy Objects
  - Process Watchers
  - Custom Policies
- Platform Support
  - All Platforms
  - Windows
    - Subprocess Support on Windows
  - macOS
- High-level API Index
  - Tasks
- Queues
- Subprocesses
- Streams
- Synchronization
- Exceptions
- Low-level API Index
  - Obtaining the Event Loop
  - Event Loop Methods
  - Transports
  - Protocols
  - Event Loop Policies
- Developing with asyncio
  - Debug Mode
  - Concurrency and Multithreading
  - Running Blocking Code
  - Logging
  - Detect never-awaited coroutines
  - Detect never-retrieved exceptions
- socket — Low-level networking interface
  - Socket families
  - Module contents
    - Exceptions
    - Constants
    - Functions
      - Creating sockets
      - Other functions
- Socket Objects
- Notes on socket timeouts
  - Timeouts and the connect method
  - Timeouts and the accept method
- Example
- ssl — TLS/SSL wrapper for socket objects
  - Functions, Constants, and Exceptions
  - Socket creation
- Context creation
- Exceptions
- Random generation
- Certificate handling
- Constants
- SSL Sockets
- SSL Contexts
- Certificates
  - Certificate chains
  - CA certificates
  - Combined key and certificate
  - Self-signed certificates
- Examples
  - Testing for SSL support
  - Client-side operation
  - Server-side operation
- Notes on non-blocking sockets
- Memory BIO Support
- SSL session
- Security considerations
  - Best defaults
  - Manual settings
    - Verifying certificates
    - Protocol versions
    - Cipher selection
  - Multi-processing
- TLS 1.3
- LibreSSL support
- `select` — Waiting for I/O completion
  - `/dev/poll` Polling Objects
  - Edge and Level Trigger Polling (epoll) Objects
  - Polling Objects
  - Kqueue Objects
  - Kevent Objects
- **selectors** — High-level I/O multiplexing
  - Introduction
  - Classes
  - Examples
- **asyncore** — Asynchronous socket handler
  - asyncore Example basic HTTP client
  - asyncore Example basic echo server
- **asynchat** — Asynchronous socket command/response handler
  - asynchat Example
- **signal** — Set handlers for asynchronous events
  - General rules
    - Execution of Python signal handlers
    - Signals and threads
  - Module contents
  - Example
  - Note on SIGPIPE
- **mmap** — Memory-mapped file support
- **Internet Data Handling**
  - **email** — An email and MIME handling package
    - **email.message**: Representing an email message
    - **email.parser**: Parsing email messages
      - FeedParser API
      - Parser API
      - Additional notes
    - **email.generator**: Generating MIME documents
    - **email.policy**: Policy Objects
    - **email.errors**: Exception and Defect classes
    - **email.headerregistry**: Custom Header Objects
    - **email.contentmanager**: Managing MIME Content
      - Content Manager Instances
- **email**: Examples
- **email.message.Message**: Representing an email message using the compat32 API
- **email.mime**: Creating email and MIME objects from scratch
- **email.header**: Internationalized headers
- **email.charset**: Representing character sets
- **email.encoders**: Encoders
- **email.utils**: Miscellaneous utilities
- **email.iterators**: Iterators
- **json**: JSON encoder and decoder
  - Basic Usage
  - Encoders and Decoders
  - Exceptions
  - Standard Compliance and Interoperability
    - Character Encodings
    - Infinite and NaN Number Values
    - Repeated Names Within an Object
    - Top-level Non-Object, Non-Array Values
    - Implementation Limitations
  - Command Line Interface
    - Command line options
- **mailcap**: Mailcap file handling
- **mailbox**: Manipulate mailboxes in various formats
  - Mailbox objects
    - Maildir
    - mbox
    - MH
    - Babyl
    - MMDF
  - Message objects
    - MaildirMessage
    - mboxMessage
- MHMessage
- BabylMessage
- MMDFMessage

- Exceptions
- Examples

- `mimetypes` — Map filenames to MIME types
  - MimeTypes Objects

- `base64` — Base16, Base32, Base64, Base85 Data Encodings

- `binhex` — Encode and decode binhex4 files

- Notes

- `binascii` — Convert between binary and ASCII

- `quopri` — Encode and decode MIME quoted-printable data

- `uu` — Encode and decode uuencode files

- **Structured Markup Processing Tools**

  - `html` — HyperText Markup Language support
  - `html.parser` — Simple HTML and XHTML parser
    - Example HTML Parser Application
    - HTMLParser Methods
    - Examples

  - `html.entities` — Definitions of HTML general entities

- **XML Processing Modules**

  - XML vulnerabilities

  - The `defusedxml` and `defusedexpat` Packages

- `xml.etree.ElementTree` — The ElementTree XML API

  - Tutorial
    - XML tree and elements
    - Parsing XML
    - Pull API for non-blocking parsing
    - Finding interesting elements
    - Modifying an XML File
- Building XML documents
- Parsing XML with Namespaces
- Additional resources
- XPath support
  - Example
  - Supported XPath syntax
- Reference
  - Functions
  - Element Objects
  - ElementTree Objects
  - QName Objects
  - TreeBuilder Objects
  - XMLParser Objects
  - XMLPullParser Objects
  - Exceptions
- `xml.dom — The Document Object Model API`
  - Module Contents
    - Objects in the DOM
      - DOMImplementation Objects
      - Node Objects
      - NodeList Objects
      - DocumentType Objects
      - Document Objects
      - Element Objects
      - Attr Objects
      - NamedNodeMap Objects
      - Comment Objects
      - Text and CDATASection Objects
      - ProcessingInstruction Objects
      - Exceptions
    - Conformance
      - Type Mapping
      - Accessor Methods
- `xml.dom.minidom — Minimal DOM implementation`
- DOM Objects
  - DOM Example
    - minidom and the DOM standard
  - `xml.dom.pulldom` — Support for building partial DOM trees
    - DOMEventStream Objects
  - `xml.sax` — Support for SAX2 parsers
    - SAXException Objects
  - `xml.sax.handler` — Base classes for SAX handlers
    - ContentHandler Objects
    - DTDHandler Objects
    - EntityResolver Objects
    - ErrorHandler Objects
  - `xml.sax.saxutils` — SAX Utilities
  - `xml.sax.xmlreader` — Interface for XML parsers
    - XMLReader Objects
    - IncrementalParser Objects
    - Locator Objects
    - InputSource Objects
    - The Attributes Interface
    - The AttributesNS Interface
  - `xml.parsers.expat` — Fast XML parsing using Expat
    - XMLParser Objects
    - ExpatError Exceptions
    - Example
    - Content Model Descriptions
    - Expat error constants
- Internet Protocols and Support
  - `webbrowser` — Convenient Web-browser controller
    - Browser Controller Objects
  - `cgi` — Common Gateway Interface support
    - Introduction
    - Using the cgi module
- Higher Level Interface
- Functions
- Caring about security
- Installing your CGI script on a Unix system
- Testing your CGI script
- Debugging CGI scripts
- Common problems and solutions

- cgitb — Traceback manager for CGI scripts
- wsgiref — WSGI Utilities and Reference Implementation
  - wsgiref.util — WSGI environment utilities
  - wsgiref.headers — WSGI response header tools
  - wsgiref.simple_server — a simple WSGI HTTP server
  - wsgiref.validate — WSGI conformance checker
  - wsgiref.handlers — server/gateway base classes
- Examples
- urllib — URL handling modules
- urllib.request — Extensible library for opening URLs
  - Request Objects
  - OpenerDirector Objects
  - BaseHandler Objects
  - HTTPRedirectHandler Objects
  - HTTPCookieProcessor Objects
  - ProxyHandler Objects
  - HTTPPasswordMgr Objects
  - HTTPPasswordMgrWithPriorAuth Objects
  - AbstractBasicAuthHandler Objects
  - HTTPBasicAuthHandler Objects
  - ProxyBasicAuthHandler Objects
- AbstractDigestAuthHandler Objects
- HTTPDigestAuthHandler Objects
- ProxyDigestAuthHandler Objects
- HTTPHandler Objects
- HTTPSHandler Objects
- FileHandler Objects
- DataHandler Objects
- FTPHandler Objects
- CacheFTPHandler Objects
- UnknownHandler Objects
- HTTPErrorProcessor Objects
- Examples
- Legacy interface
- urllib.request Restrictions
- urllib.response — Response classes used by urllib
- urllib.parse — Parse URLs into components
  - URL Parsing
  - Parsing ASCII Encoded Bytes
  - Structured Parse Results
  - URL Quoting
- urllib.error — Exception classes raised by urllib.request
- urllib.robotparser — Parser for robots.txt
- http — HTTP modules
  - HTTP status codes
- http.client — HTTP protocol client
  - HTTPConnection Objects
  - HTTPResponse Objects
  - Examples
  - HTTPMessage Objects
- ftplib — FTP protocol client
  - FTP Objects
  - FTP_TLS Objects
- **poplib** — POP3 protocol client
  - POP3 Objects
  - POP3 Example
- **imaplib** — IMAP4 protocol client
  - IMAP4 Objects
  - IMAP4 Example
- **nntplib** — NNTP protocol client
  - NNTP Objects
    - Attributes
    - Methods
    - Utility functions
- **smtplib** — SMTP protocol client
  - SMTP Objects
  - SMTP Example
- **smtpd** — SMTP Server
  - SMTPServer Objects
  - DebuggingServer Objects
  - PureProxy Objects
  - MailmanProxy Objects
  - SMTPChannel Objects
- **telnetlib** — Telnet client
  - Telnet Objects
  - Telnet Example
- **uuid** — UUID objects according to **RFC 4122**
  - Example
- **socketserver** — A framework for network servers
  - Server Creation Notes
  - Server Objects
  - Request Handler Objects
  - Examples
    - `socketserver.TCPServer` Example
    - `socketserver.UDPServer` Example
    - Asynchronous Mixins
- **http.server** — HTTP servers
- **http.cookies** — HTTP state management
  - Cookie Objects
  - Morsel Objects
  - Example
- **http.cookiejar** — Cookie handling for HTTP clients
  - CookieJar and FileCookieJar Objects
  - FileCookieJar subclasses and co-operation with web browsers
  - CookiePolicy Objects
  - DefaultCookiePolicy Objects
  - Cookie Objects
  - Examples
- **xmlrpc** — XMLRPC server and client modules
- **xmlrpc.client** — XML-RPC client access
  - ServerProxy Objects
  - DateTime Objects
  - Binary Objects
  - Fault Objects
  - ProtocolError Objects
  - MultiCall Objects
  - Convenience Functions
  - Example of Client Usage
  - Example of Client and Server Usage
- **xmlrpc.server** — Basic XML-RPC servers
  - SimpleXMLRPCServer Objects
    - SimpleXMLRPCServer Example
  - CGIXMLRPCRequestHandler
  - Documenting XMLRPC server
  - DocXMLRPCServer Objects
  - DocCGIXMLRPCRequestHandler
- **ipaddress** — IPv4/IPv6 manipulation library
  - Convenience factory functions
  - IP Addresses
- Address objects
- Conversion to Strings and Integers
- Operators
  - Comparison operators
  - Arithmetic operators
- IP Network definitions
  - Prefix, net mask and host mask
  - Network objects
  - Operators
    - Logical operators
    - Iteration
    - Networks as containers of addresses
- Interface objects
  - Operators
    - Logical operators
- Other Module Level Functions
  - Custom Exceptions
- Multimedia Services
  - audioop — Manipulate raw audio data
  - aifc — Read and write AIFF and AIFC files
  - sunau — Read and write Sun AU files
    - AU_read Objects
    - AU_write Objects
  - wave — Read and write WAV files
    - Wave_read Objects
    - Wave_write Objects
  - chunk — Read IFF chunked data
  - colorsys — Conversions between color systems
  - imghdr — Determine the type of an image
  - sndhdr — Determine type of sound file
  - ossaudiodev — Access to OSS-compatible audio devices
    - Audio Device Objects
Mixer Device Objects

- Internationalization
  - gettext — Multilingual internationalization services
  - GNU gettext API
  - Class-based API
    - The NullTranslations class
    - The GNUTranslations class
    - Solaris message catalog support
    - The Catalog constructor
  - Internationalizing your programs and modules
    - Localizing your module
    - Localizing your application
    - Changing languages on the fly
    - Deferred translations
  - Acknowledgements
    - locale — Internationalization services
      - Background, details, hints, tips and caveats
      - For extension writers and programs that embed Python
      - Access to message catalogs

- Program Frameworks
  - turtle — Turtle graphics
    - Introduction
    - Overview of available Turtle and Screen methods
      - Turtle methods
      - Methods of TurtleScreen/Screen
    - Methods of RawTurtle/Turtle and corresponding functions
      - Turtle motion
      - Tell Turtle’s state
      - Settings for measurement
      - Pen control
        - Drawing state
        - Color control
- Filling
  - More drawing control
- Turtle state
  - Visibility
  - Appearance
- Using events
- Special Turtle methods
- Compound shapes
- Methods of TurtleScreen/Screen and corresponding functions
  - Window control
  - Animation control
  - Using screen events
  - Input methods
  - Settings and special methods
  - Methods specific to Screen, not inherited from TurtleScreen
- Public classes
- Help and configuration
  - How to use help
  - Translation of docstrings into different languages
  - How to configure Screen and Turtles
- turtledemo — Demo scripts
- Changes since Python 2.6
- Changes since Python 3.0
- cmd — Support for line-oriented command interpreters
  - Cmd Objects
  - Cmd Example
- shlex — Simple lexical analysis
  - shlex Objects
  - Parsing Rules
  - Improved Compatibility with Shells
- Graphical User Interfaces with Tk
- tkinter — Python interface to Tcl/Tk
  - Tkinter Modules
  - Tkinter Life Preserver
    - How To Use This Section
    - A Simple Hello World Program
  - A (Very) Quick Look at Tcl/Tk
  - Mapping Basic Tk into Tkinter
  - How Tk and Tkinter are Related
  - Handy Reference
    - Setting Options
    - The Packer
    - Packer Options
    - Coupling Widget Variables
    - The Window Manager
    - Tk Option Data Types
    - Bindings and Events
    - The index Parameter
    - Images
  - File Handlers
- tkinter.ttk — Tk themed widgets
  - Using Ttk
  - Ttk Widgets
  - Widget
    - Standard Options
    - Scrollable Widget Options
    - Label Options
    - Compatibility Options
    - Widget States
    - ttk.Widget
  - Combobox
    - Options
    - Virtual events
    - ttk.Combobox
  - Spinbox
- Options
- Virtual events
- ttk.Spinbox
- Notebook
  - Options
  - Tab Options
  - Tab Identifiers
  - Virtual Events
  - ttk.Notebook
- Progressbar
  - Options
  - ttk.Progressbar
- Separator
  - Options
- Sizegrip
  - Platform-specific notes
  - Bugs
- Treeview
  - Options
  - Item Options
  - Tag Options
  - Column Identifiers
  - Virtual Events
  - ttk.Treeview
- Ttk Styling
  - Layouts
- `tkinter.tix` — Extension widgets for Tk
  - Using Tix
  - Tix Widgets
    - Basic Widgets
    - File Selectors
    - Hierarchical ListBox
    - Tabular ListBox
    - Manager Widgets
- Image Types
- Miscellaneous Widgets
- Form Geometry Manager
- Tix Commands
  - `tkinter.scrolledtext` — Scrolled Text Widget
- IDLE
  - Menus
    - File menu (Shell and Editor)
    - Edit menu (Shell and Editor)
    - Format menu (Editor window only)
    - Run menu (Editor window only)
    - Shell menu (Shell window only)
    - Debug menu (Shell window only)
    - Options menu (Shell and Editor)
    - Window menu (Shell and Editor)
    - Help menu (Shell and Editor)
  - Context Menus
  - Editing and navigation
    - Editor windows
    - Key bindings
    - Automatic indentation
    - Completions
    - Calltips
    - Code Context
    - Python Shell window
    - Text colors
  - Startup and code execution
    - Command line usage
    - Startup failure
    - Running user code
    - User output in Shell
    - Developing tkinter applications
    - Running without a subprocess
  - Help and preferences
- Help sources
- Setting preferences
- IDLE on macOS
- Extensions
- Other Graphical User Interface Packages

- Development Tools
  - typing — Support for type hints
    - Type aliases
    - NewType
    - Callable
    - Generics
    - User-defined generic types
    - The Any type
    - Classes, functions, and decorators
  - pydoc — Documentation generator and online help system
  - doctest — Test interactive Python examples
    - Simple Usage: Checking Examples in Docstrings
    - Simple Usage: Checking Examples in a Text File
    - How It Works
      - Which Docstrings Are Examined?
      - How are Docstring Examples Recognized?
      - What’s the Execution Context?
      - What About Exceptions?
      - Option Flags
      - Directives
      - Warnings
  - Basic API
  - Unittest API
  - Advanced API
    - DocTest Objects
    - Example Objects
    - DocTestFinder objects
    - DocTestParser objects
- DocTestRunner objects
- OutputChecker objects
- Debugging
- Soapbox

**unittest** — Unit testing framework
- Basic example
- Command-Line Interface
  - Command-line options
- Test Discovery
- Organizing test code
- Re-using old test code
- Skipping tests and expected failures
- Distinguishing test iterations using subtests
- Classes and functions
  - Test cases
    - Deprecated aliases
  - Grouping tests
  - Loading and running tests
    - `load_tests` Protocol
- Class and Module Fixtures
  - `setUpClass` and `tearDownClass`
  - `setUpModule` and `tearDownModule`
- Signal Handling

**unittest.mock** — mock object library
- Quick Guide
- The Mock Class
  - Calling
  - Deleting Attributes
  - Mock names and the name attribute
  - Attaching Mocks as Attributes
- The patchers
  - `patch`
  - `patch.object`
  - `patch.dict`
patch.multiple
patch methods: start and stop
patch builtins
TEST_PREFIX
Nesting Patch Decorators
Where to patch
Patching Descriptors and Proxy Objects
  MagicMock and magic method support
    Mocking Magic Methods
    Magic Mock
  Helpers
    sentinel
    DEFAULT
call
create_autospec
ANY
FILTER_DIR
mock_open
Autospeccing
Sealing mocks

unittest.mock — getting started
  Using Mock
    Mock Patching Methods
    Mock for Method Calls on an Object
    Mocking Classes
    Naming your mocks
    Tracking all Calls
    Setting Return Values and Attributes
    Raising exceptions with mocks
    Side effect functions and iterables
    Creating a Mock from an Existing Object
  Patch Decorators
  Further Examples
    Mocking chained calls
- Partial mocking
- Mocking a Generator Method
- Applying the same patch to every test method
- Mocking Unbound Methods
- Checking multiple calls with mock
- Coping with mutable arguments
- Nesting Patches
- Mocking a dictionary with MagicMock
- Mock subclasses and their attributes
- Mocking imports with patch.dict
- Tracking order of calls and less verbose call assertions
- More complex argument matching

- **2to3** - Automated Python 2 to 3 code translation
  - Using 2to3
  - Fixers
  - *lib2to3* - 2to3's library

- **test** — Regression tests package for Python
  - Writing Unit Tests for the test package
  - Running tests using the command-line interface

- **test.support** — Utilities for the Python test suite
- **test.support.script_helper** — Utilities for the Python execution tests

**Debugging and Profiling**
- **bdb** — Debugger framework
- **faulthandler** — Dump the Python traceback
  - Dumping the traceback
  - Fault handler state
  - Dumping the tracebacks after a timeout
  - Dumping the traceback on a user signal
  - Issue with file descriptors
  - Example
- **pdb** — The Python Debugger
  - Debugger Commands
- The Python Profilers
  - Introduction to the profilers
  - profile and cProfile Module Reference
  - The Stats Class
  - What Is Deterministic Profiling?
  - Limitations
  - Calibration
  - Using a custom timer
- timeit — Measure execution time of small code snippets
  - Basic Examples
  - Python Interface
  - Command-Line Interface
  - Examples
- trace — Trace or track Python statement execution
  - Command-Line Usage
    - Main options
    - Modifiers
    - Filters
  - Programmatic Interface
- tracemalloc — Trace memory allocations
  - Examples
    - Display the top 10
    - Compute differences
    - Get the traceback of a memory block
    - Pretty top
  - API
    - Functions
    - DomainFilter
    - Filter
    - Frame
    - Snapshot
    - Statistic
- StatisticDiff
- Trace
- Traceback

**Software Packaging and Distribution**
- distutils — Building and installing Python modules
- ensurepip — Bootstrapping the pip installer
  - Command line interface
  - Module API
- venv — Creation of virtual environments
  - Creating virtual environments
  - API
  - An example of extending EnvBuilder
- zipapp — Manage executable Python zip archives
  - Basic Example
  - Command-Line Interface
  - Python API
  - Examples
  - Specifying the Interpreter
  - Creating Standalone Applications with zipapp
    - Making a Windows executable
  - Caveats
  - The Python Zip Application Archive Format

**Python Runtime Services**
- sys — System-specific parameters and functions
- sysconfig — Provide access to Python's configuration information
  - Configuration variables
  - Installation paths
  - Other functions
  - Using sysconfig as a script
- builtins — Built-in objects
- __main__ — Top-level script environment
- warnings — Warning control
- Warning Categories
- The Warnings Filter
  - Describing Warning Filters
  - Default Warning Filter
  - Overriding the default filter
- Temporarily Suppressing Warnings
- Testing Warnings
- Updating Code For New Versions of Dependencies
- Available Functions
- Available Context Managers
- `dataclasses` — Data Classes
  - Module-level decorators, classes, and functions
  - Post-init processing
  - Class variables
  - Init-only variables
  - Frozen instances
  - Inheritance
  - Default factory functions
  - Mutable default values
  - Exceptions
- `contextlib` — Utilities for `with`-statement contexts
  - Utilities
  - Examples and Recipes
    - Supporting a variable number of context managers
    - Catching exceptions from `__enter__` methods
    - Cleaning up in an `__enter__` implementation
    - Replacing any use of `try-finally` and flag variables
    - Using a context manager as a function decorator
    - Single use, reusable and reentrant context managers
    - Reentrant context managers
- Reusable context managers
  - abc — Abstract Base Classes
  - atexit — Exit handlers
    - atexit Example
  - traceback — Print or retrieve a stack traceback
    - TracebackException Objects
    - StackSummary Objects
    - FrameSummary Objects
    - Traceback Examples
  - __future__ — Future statement definitions
  - gc — Garbage Collector interface
  - inspect — Inspect live objects
    - Types and members
    - Retrieving source code
    - Introspecting callables with the Signature object
    - Classes and functions
    - The interpreter stack
    - Fetching attributes statically
    - Current State of Generators and Coroutines
    - Code Objects Bit Flags
    - Command Line Interface
  - site — Site-specific configuration hook
    - Readline configuration
    - Module contents
  - Custom Python Interpreters
    - code — Interpreter base classes
      - Interactive Interpreter Objects
      - Interactive Console Objects
  - codeop — Compile Python code
  - Importing Modules
    - zipimport — Import modules from Zip archives
      - zipimporter Objects
      - Examples
- pkgutil — Package extension utility
- modulefinder — Find modules used by a script
  - Example usage of ModuleFinder
- runpy — Locating and executing Python modules
- importlib — The implementation of import
  - Introduction
  - Functions
  - importlib.abc — Abstract base classes related to import
  - importlib.resources — Resources
  - importlib.machinery — Importers and path hooks
  - importlib.util — Utility code for importers
  - Examples
    - Importing programmatically
    - Checking if a module can be imported
    - Importing a source file directly
    - Setting up an importer
    - Approximating
      importlib.import_module()
- Python Language Services
  - parser — Access Python parse trees
    - Creating ST Objects
    - Converting ST Objects
    - Queries on ST Objects
    - Exceptions and Error Handling
    - ST Objects
    - Example: Emulation of compile()
  - ast — Abstract Syntax Trees
    - Node classes
    - Abstract Grammar
    - ast Helpers
  - symtable — Access to the compiler's symbol tables
Generating Symbol Tables
Examining Symbol Tables
- `symbol` — Constants used with Python parse trees
- `token` — Constants used with Python parse trees
- `keyword` — Testing for Python keywords
- `tokenize` — Tokenizer for Python source
  - Tokenizing Input
  - Command-Line Usage
  - Examples
- `tabnanny` — Detection of ambiguous indentation
- `pyclbr` — Python class browser support
  - Function Objects
  - Class Objects
- `py_compile` — Compile Python source files
- `compileall` — Byte-compile Python libraries
  - Command-line use
  - Public functions
- `dis` — Disassembler for Python bytecode
  - Bytecode analysis
  - Analysis functions
  - Python Bytecode Instructions
  - Opcode collections
- `pickletools` — Tools for pickle developers
  - Command line usage
    - Command line options
    - Programmatic Interface
- Miscellaneous Services
  - `formatter` — Generic output formatting
    - The Formatter Interface
    - Formatter Implementations
    - The Writer Interface
    - Writer Implementations
- MS Windows Specific Services
- **msilib** — Read and write Microsoft Installer files
  - Database Objects
  - View Objects
  - Summary Information Objects
  - Record Objects
  - Errors
  - CAB Objects
  - Directory Objects
  - Features
  - GUI classes
  - Precomputed tables
- **msvcrt** — Useful routines from the MS VC++ runtime
  - File Operations
  - Console I/O
  - Other Functions
- **winreg** — Windows registry access
  - Functions
  - Constants
    - HKEY_* Constants
    - Access Rights
      - 64-bit Specific
    - Value Types
  - Registry Handle Objects
- **winsound** — Sound-playing interface for Windows
- **Unix Specific Services**
  - **posix** — The most common POSIX system calls
    - Large File Support
    - Notable Module Contents
  - **pwd** — The password database
  - **spwd** — The shadow password database
  - **grp** — The group database
  - **crypt** — Function to check Unix passwords
    - Hashing Methods
- Module Attributes
- Module Functions
- Examples
  - termios — POSIX style tty control
    - Example
  - tty — Terminal control functions
  - pty — Pseudo-terminal utilities
    - Example
  - fcntl — The fcntl and ioctl system calls
  - pipes — Interface to shell pipelines
    - Template Objects
  - resource — Resource usage information
    - Resource Limits
    - Resource Usage
  - nis — Interface to Sun's NIS (Yellow Pages)
  - syslog — Unix syslog library routines
    - Examples
      - Simple example
- Superseded Modules
  - optparse — Parser for command line options
    - Background
      - Terminology
      - What are options for?
      - What are positional arguments for?
    - Tutorial
      - Understanding option actions
      - The store action
      - Handling boolean (flag) options
      - Other actions
      - Default values
      - Generating help
        - Grouping Options
      - Printing a version string
- How optparse handles errors
- Putting it all together

**Reference Guide**
- Creating the parser
- Populating the parser
- Defining options
- Option attributes
- Standard option actions
- Standard option types
- Parsing arguments
- Querying and manipulating your option parser
- Conflicts between options
- Cleanup
- Other methods

**Option Callbacks**
- Defining a callback option
- How callbacks are called
- Raising errors in a callback
- Callback example 1: trivial callback
- Callback example 2: check option order
- Callback example 3: check option order (generalized)
- Callback example 4: check arbitrary condition
- Callback example 5: fixed arguments
- Callback example 6: variable arguments

**Extending optparse**
- Adding new types
- Adding new actions

- **imp** — Access the import internals

**Examples**
- Undocumented Modules
  - Platform specific modules
- Extending and Embedding the Python Interpreter
  - Recommended third party tools
Creating extensions without third party tools

1. Extending Python with C or C++
   1.1. A Simple Example
   1.2. Intermezzo: Errors and Exceptions
   1.3. Back to the Example
   1.4. The Module’s Method Table and Initialization Function
   1.5. Compilation and Linkage
   1.6. Calling Python Functions from C
   1.7. Extracting Parameters in Extension Functions
   1.8. Keyword Parameters for Extension Functions
   1.9. Building Arbitrary Values
   1.10. Reference Counts
      1.10.1. Reference Counting in Python
      1.10.2. Ownership Rules
      1.10.3. Thin Ice
      1.10.4. NULL Pointers
   1.11. Writing Extensions in C++
   1.12. Providing a C API for an Extension Module

2. Defining Extension Types: Tutorial
   2.1. The Basics
   2.2. Adding data and methods to the Basic example
   2.3. Providing finer control over data attributes
   2.4. Supporting cyclic garbage collection
   2.5. Subclassing other types

3. Defining Extension Types: Assorted Topics
   3.1. Finalization and De-allocation
   3.2. Object Presentation
   3.3. Attribute Management
      3.3.1. Generic Attribute Management
      3.3.2. Type-specific Attribute Management
   3.4. Object Comparison
   3.5. Abstract Protocol Support
   3.6. Weak Reference Support
3.7. More Suggestions
4. Building C and C++ Extensions
   4.1. Building C and C++ Extensions with distutils
   4.2. Distributing your extension modules
5. Building C and C++ Extensions on Windows
   5.1. A Cookbook Approach
   5.2. Differences Between Unix and Windows
   5.3. Using DLLs in Practice
   Embedding the CPython runtime in a larger application
   1. Embedding Python in Another Application
      1.1. Very High Level Embedding
      1.2. Beyond Very High Level Embedding: An overview
      1.3. Pure Embedding
      1.4. Extending Embedded Python
      1.5. Embedding Python in C++
      1.6. Compiling and Linking under Unix-like systems

- Python/C API Reference Manual
  - Introduction
    - Coding standards
    - Include Files
    - Useful macros
    - Objects, Types and Reference Counts
      - Reference Counts
        - Reference Count Details
    - Types
    - Exceptions
    - Embedding Python
    - Debugging Builds
      - Stable Application Binary Interface
      - The Very High Level Layer
      - Reference Counting
      - Exception Handling
        - Printing and clearing
- Raising exceptions
- Issuing warnings
- Querying the error indicator
- Signal Handling
- Exception Classes
- Exception Objects
- Unicode Exception Objects
- Recursion Control
- Standard Exceptions
- Standard Warning Categories

- Utilities
  - Operating System Utilities
  - System Functions
  - Process Control
  - Importing Modules
  - Data marshalling support
  - Parsing arguments and building values
    - Parsing arguments
      - Strings and buffers
      - Numbers
      - Other objects
    - API Functions
  - Building values
  - String conversion and formatting
- Reflection
- Codec registry and support functions
  - Codec lookup API
  - Registry API for Unicode encoding error handlers

- Abstract Objects Layer
  - Object Protocol
  - Number Protocol
  - Sequence Protocol
  - Mapping Protocol
  - Iterator Protocol
- Buffer Protocol
  - Buffer structure
  - Buffer request types
    - request-independent fields
    - readonly, format
    - shape, strides, suboffsets
    - contiguity requests
    - compound requests
  - Complex arrays
    - NumPy-style: shape and strides
    - PIL-style: shape, strides and suboffsets
  - Buffer-related functions
- Old Buffer Protocol
  - Concrete Objects Layer
    - Fundamental Objects
      - Type Objects
      - The None Object
    - Numeric Objects
      - Integer Objects
      - Boolean Objects
      - Floating Point Objects
      - Complex Number Objects
        - Complex Numbers as C Structures
        - Complex Numbers as Python Objects
    - Sequence Objects
      - Bytes Objects
      - Byte Array Objects
        - Type check macros
        - Direct API functions
        - Macros
      - Unicode Objects and Codecs
        - Unicode Objects
          - Unicode Type
          - Unicode Character Properties
- Creating and accessing Unicode strings
- Deprecated Py_UNICODE APIs
- Locale Encoding
- File System Encoding
- wchar_t Support
- **Built-in Codecs**
  - Generic Codecs
  - UTF-8 Codecs
  - UTF-32 Codecs
  - UTF-16 Codecs
  - UTF-7 Codecs
  - Unicode-Escape Codecs
  - Raw-Unicode-Escape Codecs
  - Latin-1 Codecs
  - ASCII Codecs
  - Character Map Codecs
  - MBCS codecs for Windows
- **Methods & Slots**
- **Methods and Slot Functions**
  - Tuple Objects
  - Struct Sequence Objects
  - List Objects
- **Container Objects**
  - Dictionary Objects
  - Set Objects
- **Function Objects**
  - Function Objects
  - Instance Method Objects
  - Method Objects
  - Cell Objects
  - Code Objects
- **Other Objects**
  - File Objects
  - Module Objects
- Initializing C modules
  - Single-phase initialization
  - Multi-phase initialization
  - Low-level module creation functions
  - Support functions
- Module lookup
  - Iterator Objects
  - Descriptor Objects
  - Slice Objects
  - Ellipsis Object
  - MemoryView objects
  - Weak Reference Objects
  - Capsules
  - Generator Objects
  - Coroutine Objects
  - Context Variables Objects
  - DateTime Objects
- Initialization, Finalization, and Threads
  - Before Python Initialization
  - Global configuration variables
  - Initializing and finalizing the interpreter
  - Process-wide parameters
  - Thread State and the Global Interpreter Lock
    - Releasing the GIL from extension code
    - Non-Python created threads
    - High-level API
    - Low-level API
  - Sub-interpreter support
    - Bugs and caveats
  - Asynchronous Notifications
  - Profiling and Tracing
  - Advanced Debugger Support
  - Thread Local Storage Support
    - Thread Specific Storage (TSS) API
Dynamic Allocation
  Methods
    Thread Local Storage (TLS) API
    Memory Management
      Overview
      Raw Memory Interface
      Memory Interface
      Object allocators
      Default Memory Allocators
      Customize Memory Allocators
      The pymalloc allocator
        Customize pymalloc Arena Allocator
      tracemalloc C API
      Examples
    Object Implementation Support
      Allocating Objects on the Heap
      Common Object Structures
      Type Objects
      Number Object Structures
      Mapping Object Structures
      Sequence Object Structures
      Buffer Object Structures
      Async Object Structures
      Supporting Cyclic Garbage Collection
    API and ABI Versioning
  Distributing Python Modules
    Key terms
    Open source licensing and collaboration
    Installing the tools
    Reading the guide
    How do I…?
      ... choose a name for my project?
      ... create and distribute binary extensions?
  Installing Python Modules
Key terms

Basic usage

How do I ...?
- ... install pip in versions of Python prior to Python 3.4?
- ... install packages just for the current user?
- ... install scientific Python packages?
- ... work with multiple versions of Python installed in parallel?

Common installation issues
- Installing into the system Python on Linux
- Pip not installed
- Installing binary extensions

Python HOWTOs
- Porting Python 2 Code to Python 3
  - The Short Explanation
  - Details
    - Drop support for Python 2.6 and older
    - Make sure you specify the proper version support in your setup.py file
    - Have good test coverage
    - Learn the differences between Python 2 & 3
    - Update your code
      - Division
      - Text versus binary data
      - Use feature detection instead of version detection
    - Prevent compatibility regressions
    - Check which dependencies block your transition
    - Update your setup.py file to denote Python 3 compatibility
    - Use continuous integration to stay compatible
    - Consider using optional static type checking
- Porting Extension Modules to Python 3
  - Conditional compilation
- Changes to Object APIs
  - str/unicode Unification
  - long/int Unification
  - Module initialization and state
  - CObject replaced with Capsule
  - Other options
○ Curses Programming with Python
  - What is curses?
    - The Python curses module
  - Starting and ending a curses application
  - Windows and Pads
  - Displaying Text
    - Attributes and Color
  - User Input
  - For More Information
○ Descriptor HowTo Guide
  - Abstract
  - Definition and Introduction
  - Descriptor Protocol
  - Invoking Descriptors
  - Descriptor Example
  - Properties
  - Functions and Methods
  - Static Methods and Class Methods
○ Functional Programming HOWTO
  - Introduction
    - Formal provability
    - Modularity
    - Ease of debugging and testing
    - Composability
  - Iterators
    - Data Types That Support Iterators
  - Generator expressions and list comprehensions
  - Generators
- Passing values into a generator
- Built-in functions
- The itertools module
  - Creating new iterators
  - Calling functions on elements
  - Selecting elements
  - Combinatoric functions
  - Grouping elements
- The functools module
  - The operator module
- Small functions and the lambda expression
- Revision History and Acknowledgements
- References
  - General
  - Python-specific
  - Python documentation
- Logging HOWTO
  - Basic Logging Tutorial
    - When to use logging
    - A simple example
    - Logging to a file
    - Logging from multiple modules
    - Logging variable data
    - Changing the format of displayed messages
    - Displaying the date/time in messages
    - Next Steps
  - Advanced Logging Tutorial
    - Logging Flow
    - Loggers
    - Handlers
    - Formatters
    - Configuring Logging
    - What happens if no configuration is provided
    - Configuring Logging for a Library
- Logging Levels
  - Custom Levels
- Useful Handlers
- Exceptions raised during logging
- Using arbitrary objects as messages
- Optimization
- Logging Cookbook
  - Using logging in multiple modules
  - Logging from multiple threads
  - Multiple handlers and formatters
  - Logging to multiple destinations
  - Configuration server example
  - Dealing with handlers that block
  - Sending and receiving logging events across a network
  - Adding contextual information to your logging output
    - Using LoggerAdapters to impart contextual information
    - Using objects other than dicts to pass contextual information
    - Using Filters to impart contextual information
  - Logging to a single file from multiple processes
- Using file rotation
- Use of alternative formatting styles
- Customizing LogRecord
- Subclassing QueueHandler - a ZeroMQ example
- Subclassing QueueListener - a ZeroMQ example
- An example dictionary-based configuration
- Using a rotator and namer to customize log rotation processing
- A more elaborate multiprocessing example
- Inserting a BOM into messages sent to a SysLogHandler
- Implementing structured logging
- Customizing handlers with dictConfig()
application

- Using LogRecord factories
- Using custom message objects
- Configuring filters with dictConfig()
- Customized exception formatting
- Speaking logging messages
- Buffering logging messages and outputting them conditionally
- Formatting times using UTC (GMT) via configuration
- Using a context manager for selective logging

- Regular Expression HOWTO
  - Introduction
  - Simple Patterns
    - Matching Characters
    - Repeating Things
  - Using Regular Expressions
    - Compiling Regular Expressions
    - The Backslash Plague
    - Performing Matches
    - Module-Level Functions
    - Compilation Flags
  - More Pattern Power
    - More Metacharacters
    - Grouping
    - Non-capturing and Named Groups
    - Lookahead Assertions
  - Modifying Strings
    - Splitting Strings
    - Search and Replace
  - Common Problems
    - Use String Methods
    - match() versus search()
    - Greedy versus Non-Greedy
    - Using re.VERBOSE
- Feedback
  - Socket Programming HOWTO
    - Sockets
      - History
    - Creating a Socket
      - IPC
    - Using a Socket
      - Binary Data
    - Disconnecting
      - When Sockets Die
    - Non-blocking Sockets
  - Sorting HOW TO
    - Sorting Basics
    - Key Functions
    - Operator Module Functions
    - Ascending and Descending
    - Sort Stability and Complex Sorts
    - The Old Way Using Decorate-Sort-Undecorate
    - The Old Way Using the `cmp` Parameter
    - Odd and Ends
  - Unicode HOWTO
    - Introduction to Unicode
      - Definitions
      - Encodings
      - References
    - Python’s Unicode Support
      - The String Type
      - Converting to Bytes
      - Unicode Literals in Python Source Code
      - Unicode Properties
      - Comparing Strings
      - Unicode Regular Expressions
      - References
    - Reading and Writing Unicode Data
An introduction to the ipaddress module
  - Creating Address/Network/Interface objects
    - A Note on IP Versions
    - IP Host Addresses
    - Defining Networks
    - Host Interfaces
  - Inspecting Address/Network/Interface Objects
  - Networks as lists of Addresses
  - Comparisons
  - Using IP Addresses with other modules
  - Getting more detail when instance creation fails

Argument Clinic How-To
  - The Goals Of Argument Clinic
  - Basic Concepts And Usage
  - Converting Your First Function
  - Advanced Topics
    - Symbolic default values
    - Renaming the C functions and variables generated by Argument Clinic
    - Converting functions using PyArg_UnpackTuple
    - Optional Groups
    - Using real Argument Clinic converters, instead of “legacy converters”
    - Py_buffer
    - Advanced converters
    - Parameter default values
    - The NULL default value
    - Expressions specified as default values
    - Using a return converter
    - Cloning existing functions
    - Calling Python code
    - Using a “self converter”
    - Writing a custom converter
    - Writing a custom return converter
- METH_O and METH_NOARGS
- tp_new and tp_init functions
- Changing and redirecting Clinic's output
- The #ifdef trick
- Using Argument Clinic in Python files
  - Instrumenting CPython with DTrace and SystemTap
    - Enabling the static markers
    - Static DTrace probes
    - Static SystemTap markers
    - Available static markers
    - SystemTap Tapsets
    - Examples
- Python Frequently Asked Questions
  - General Python FAQ
    - General Information
    - Python in the real world
  - Programming FAQ
    - General Questions
    - Core Language
    - Numbers and strings
    - Performance
    - Sequences (Tuples/Lists)
    - Objects
    - Modules
  - Design and History FAQ
    - Why does Python use indentation for grouping of statements?
    - Why am I getting strange results with simple arithmetic operations?
    - Why are floating-point calculations so inaccurate?
    - Why are Python strings immutable?
    - Why must ‘self’ be used explicitly in method definitions and calls?
    - Why can’t I use an assignment in an expression?
- Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?  
- Why is join() a string method instead of a list or tuple method?  
- How fast are exceptions?  
- Why isn’t there a switch or case statement in Python?  
- Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?  
- Why can’t lambda expressions contain statements?  
- Can Python be compiled to machine code, C or some other language?  
- How does Python manage memory?  
- Why doesn’t CPython use a more traditional garbage collection scheme?  
- Why isn’t all memory freed when CPython exits?  
- Why are there separate tuple and list data types?  
- How are lists implemented in CPython?  
- How are dictionaries implemented in CPython?  
- Why must dictionary keys be immutable?  
- Why doesn’t list.sort() return the sorted list?  
- How do you specify and enforce an interface spec in Python?  
- Why is there no goto?  
- Why can’t raw strings (r-strings) end with a backslash?  
- Why doesn’t Python have a “with” statement for attribute assignments?  
- Why are colons required for the if/while/def/class statements?  
- Why does Python allow commas at the end of lists and tuples?  

Library and Extension FAQ
- General Library Questions  
- Common tasks  
- Threads
Input and Output
Network/Internet Programming
Databases
Mathematics and Numerics

- **Extending/Embedding FAQ**
  - Can I create my own functions in C?
  - Can I create my own functions in C++?
  - Writing C is hard; are there any alternatives?
  - How can I execute arbitrary Python statements from C?
  - How can I evaluate an arbitrary Python expression from C?
  - How do I extract C values from a Python object?
  - How do I use `Py_BuildValue()` to create a tuple of arbitrary length?
  - How do I call an object's method from C?
  - How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
  - How do I access a module written in Python from C?
  - How do I interface to C++ objects from Python?
  - I added a module using the Setup file and the make fails; why?
  - How do I debug an extension?
  - I want to compile a Python module on my Linux system, but some files are missing. Why?
  - How do I tell “incomplete input” from “invalid input”?
  - How do I find undefined g++ symbols `__builtin_new` or `__pure_virtual`?
  - Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

- **Python on Windows FAQ**
  - How do I run a Python program under Windows?
  - How do I make Python scripts executable?
  - Why does Python sometimes take so long to start?
- How do I make an executable from a Python script?
- Is a *.pyd file the same as a DLL?
- How can I embed Python into a Windows application?
- How do I keep editors from inserting tabs into my Python source?
- How do I check for a keypress without blocking?
  - Graphic User Interface FAQ
    - General GUI Questions
      - What platform-independent GUI toolkits exist for Python?
      - What platform-specific GUI toolkits exist for Python?
    - Tkinter questions
      - “Why is Python Installed on my Computer?” FAQ
        - What is Python?
        - Why is Python installed on my machine?
        - Can I delete Python?

- Glossary
- About these documents
  - Contributors to the Python Documentation
- Dealing with Bugs
  - Documentation bugs
  - Using the Python issue tracker
  - Getting started contributing to Python yourself
- Copyright
- History and License
  - History of the software
  - Terms and conditions for accessing or otherwise using Python
    - PSF LICENSE AGREEMENT FOR PYTHON 3.7.3
    - BeOpen.com LICENSE AGREEMENT FOR PYTHON 2.0
    - CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1
    - CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2
  - Licenses and Acknowledgements for Incorporated Software
- Mersenne Twister
- Sockets
- Asynchronous socket services
- Cookie management
- Execution tracing
- UUencode and UUdecode functions
- XML Remote Procedure Calls
- test_epoll
- Select kqueue
- SipHash24
- strtod and dtoa
- OpenSSL
- expat
- libffi
- zlib
- cfuhash
- libmpdec
## Python Module Index

<p>| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z |
| <strong>future</strong> | Future statement definitions |
| <strong>main</strong> | The environment where the top-level script is run. |
| _dummy_thread | Drop-in replacement for the _thread module. |
| _thread | Low-level threading API. |
| a | abc | Abstract base classes according to PEP 3119. |
| aifc | Read and write audio files in AIFF or AIFC format. |
| argparse | Command-line option and argument parsing library. |
| array | Space efficient arrays of uniformly typed numeric values. |
| ast | Abstract Syntax Tree classes and |</p>
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchat</td>
<td>Support for asynchronous command/response protocols.</td>
</tr>
<tr>
<td>asyncio</td>
<td>Asynchronous I/O.</td>
</tr>
<tr>
<td>asyncore</td>
<td>A base class for developing asynchronous socket handling services.</td>
</tr>
<tr>
<td>atexit</td>
<td>Register and execute cleanup functions.</td>
</tr>
<tr>
<td>audioop</td>
<td>Manipulate raw audio data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85</td>
</tr>
<tr>
<td>bdb</td>
<td>Debugger framework.</td>
</tr>
<tr>
<td>binascii</td>
<td>Tools for converting between binary and various ASCII-encoded binary \</td>
</tr>
<tr>
<td></td>
<td>representations.</td>
</tr>
<tr>
<td>binhex</td>
<td>Encode and decode files in binhex4 format.</td>
</tr>
<tr>
<td>bisect</td>
<td>Array bisection algorithms for binary searching.</td>
</tr>
<tr>
<td>builtins</td>
<td>The module that provides the built-in namespace.</td>
</tr>
<tr>
<td>bz2</td>
<td>Interfaces for bzip2</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>calendar</td>
<td>Functions for working with calendars, including some emulation of the Unix calendar program.</td>
</tr>
<tr>
<td>cgi</td>
<td>Helpers for running Python scripts via the Common Gateway Interface.</td>
</tr>
<tr>
<td>cgitb</td>
<td>Configurable traceback handler for CGI scripts.</td>
</tr>
<tr>
<td>chunk</td>
<td>Module to read IFF chunks.</td>
</tr>
<tr>
<td>cmath</td>
<td>Mathematical functions for complex numbers.</td>
</tr>
<tr>
<td>cmd</td>
<td>Build line-oriented command interpreters.</td>
</tr>
<tr>
<td>code</td>
<td>Facilities to implement read-eval-print loops.</td>
</tr>
<tr>
<td>codecs</td>
<td>Encode and decode data and streams.</td>
</tr>
<tr>
<td>codeop</td>
<td>Compile (possibly incomplete) Python code.</td>
</tr>
<tr>
<td>collections</td>
<td>Container datatypes</td>
</tr>
<tr>
<td></td>
<td>Abstract base classes for containers</td>
</tr>
<tr>
<td>colorsys</td>
<td>Conversion functions between RGB and other color systems.</td>
</tr>
<tr>
<td>Library</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>compileall</td>
<td>Tools for byte-compiling all Python source files in a directory tree.</td>
</tr>
<tr>
<td>concurrent</td>
<td>Execute computations concurrently using threads or processes.</td>
</tr>
<tr>
<td>concurrent.futures</td>
<td></td>
</tr>
<tr>
<td>configparser</td>
<td>Configuration file parser.</td>
</tr>
<tr>
<td>contextlib</td>
<td>Utilities for with-statement contexts.</td>
</tr>
<tr>
<td>contextvars</td>
<td>Context Variables</td>
</tr>
<tr>
<td>copy</td>
<td>Shallow and deep copy operations.</td>
</tr>
<tr>
<td>copyreg</td>
<td>Register pickle support functions.</td>
</tr>
<tr>
<td>cProfile</td>
<td>The crypt() function used to check Unix passwords.</td>
</tr>
<tr>
<td>crypt (Unix)</td>
<td></td>
</tr>
<tr>
<td>csv</td>
<td>Write and read tabular data to and from delimited files.</td>
</tr>
<tr>
<td>ctypes</td>
<td>A foreign function library for Python.</td>
</tr>
<tr>
<td>curses (Unix)</td>
<td>An interface to the curses library, providing portable terminal handling.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>curses.ascii</td>
<td>Constants and set-membership functions for ASCII characters.</td>
</tr>
<tr>
<td>curses.panel</td>
<td>A panel stack extension</td>
</tr>
</tbody>
</table>
**curses.textpad**

That adds depth to curses windows.

*Emacs-like input editing in a curses window.*

<table>
<thead>
<tr>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dataclasses</strong></td>
</tr>
<tr>
<td><strong>datetime</strong></td>
</tr>
<tr>
<td><strong>dbm</strong></td>
</tr>
<tr>
<td><strong>dbm.dumb</strong></td>
</tr>
<tr>
<td><strong>dbm.gnu (Unix)</strong></td>
</tr>
<tr>
<td><strong>dbm.ndbm (Unix)</strong></td>
</tr>
<tr>
<td><strong>difflib</strong></td>
</tr>
<tr>
<td><strong>dis</strong></td>
</tr>
<tr>
<td><strong>distutils</strong></td>
</tr>
<tr>
<td>Module</td>
</tr>
<tr>
<td>------------------------------------------</td>
</tr>
<tr>
<td>distutils.archive_util</td>
</tr>
<tr>
<td>distutils.bcppcompiler</td>
</tr>
<tr>
<td>distutils.ccompiler</td>
</tr>
<tr>
<td>distutils.cmd</td>
</tr>
<tr>
<td>distutils.command</td>
</tr>
<tr>
<td>distutils.command.bdist</td>
</tr>
<tr>
<td>distutils.command.bdist_dumb</td>
</tr>
<tr>
<td>distutils.command.bdist_msi</td>
</tr>
<tr>
<td>distutils.command.bdist_packager</td>
</tr>
<tr>
<td>distutils.command.bdist_rpm</td>
</tr>
<tr>
<td>distutils.command.bdist_wininst</td>
</tr>
<tr>
<td>distutils.command.build</td>
</tr>
</tbody>
</table>
distutils.command.build_clib
Build any C libraries in a package

distutils.command.build_ext
Build any extensions in a package

distutils.command.build_py
Build the .py/.pyc files of a package

distutils.command.build_scripts
Build the scripts of a package

distutils.command.check
Check the metadata of a package

distutils.command.clean
Clean a package build area

distutils.command.config
Perform package configuration

distutils.command.install
Install a package

distutils.command.install_data
Install data files from a package

distutils.command.install_headers
Install C/C++ header files from a package

distutils.command.install_lib
Install library files from a package

distutils.command.install_scripts
Install script files from a package

distutils.command.register
Register a module with the Python Package Index

distutils.command.sdist
Build a source distribution

distutils.core
The core Distutils functionality

distutils.cygwinccompiler

distutils.debug
Provides the debug flag
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distutils.dep_util</td>
<td>Utility functions for simple dependency checking</td>
</tr>
<tr>
<td>distutils.dir_util</td>
<td>Utility functions for operating on directories and directory trees</td>
</tr>
<tr>
<td>distutils.dist</td>
<td>Provides the Distribution class, which represents the module distribution being built/installed/distributed</td>
</tr>
<tr>
<td>distutils.errors</td>
<td>Provides standard distutils exceptions</td>
</tr>
<tr>
<td>distutils.extension</td>
<td>Provides the Extension class, used to describe C/C++ extension modules in setup scripts</td>
</tr>
<tr>
<td>distutils.fancy_getopt</td>
<td>Additional getopt functionality</td>
</tr>
<tr>
<td>distutils.file_util</td>
<td>Utility functions for operating on single files</td>
</tr>
<tr>
<td>distutils.filelist</td>
<td>The FileList class, used for poking about the file system and building lists of files.</td>
</tr>
<tr>
<td>distutils.log</td>
<td>A simple logging mechanism, 282-style</td>
</tr>
<tr>
<td>distutils.msvccompiler</td>
<td>Microsoft Compiler</td>
</tr>
<tr>
<td>distutils.spawn</td>
<td>Provides the spawn() function</td>
</tr>
<tr>
<td>distutils.sysconfig</td>
<td>Low-level access to configuration</td>
</tr>
</tbody>
</table>
distutils.text_file

provides the TextFile class, a simple interface to text files

distutils.unixccompiler

UNIX C Compiler

distutils.util

Miscellaneous other utility functions

distutils.version

implements classes that represent module version numbers.

doctest

Test pieces of code within docstrings.

dummy_threading

Drop-in replacement for the threading module.

<table>
<thead>
<tr>
<th>e</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>Package supporting the parsing, manipulating, and generating email messages.</td>
</tr>
<tr>
<td>email.charset</td>
<td>Character Sets</td>
</tr>
<tr>
<td>email.contentmanager</td>
<td>Storing and Retrieving Content from MIME Parts</td>
</tr>
<tr>
<td>email.encoders</td>
<td>Encoders for email message payloads.</td>
</tr>
<tr>
<td>email.errors</td>
<td>The exception classes used by the email package.</td>
</tr>
<tr>
<td>email.generator</td>
<td>Generate flat text email messages from a message structure.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>email.header</td>
<td>Representing non-ASCII headers</td>
</tr>
<tr>
<td>email.headerregistry</td>
<td>Automatic Parsing of headers based on the field name</td>
</tr>
<tr>
<td>email.iterators</td>
<td>Iterate over a message object tree</td>
</tr>
<tr>
<td>email.message</td>
<td>The base class representing email messages</td>
</tr>
<tr>
<td>email.mime</td>
<td>Build MIME messages</td>
</tr>
<tr>
<td>email.parser</td>
<td>Parse flat text email messages to produce a message object structure</td>
</tr>
<tr>
<td>email.policy</td>
<td>Controlling the parsing and generating of messages</td>
</tr>
<tr>
<td>email.utils</td>
<td>Miscellaneous email package utilities</td>
</tr>
<tr>
<td>encodings</td>
<td></td>
</tr>
<tr>
<td>encodings.idna</td>
<td>Internationalized Domain Names implementation</td>
</tr>
<tr>
<td>encodings.mbcfs</td>
<td>Windows ANSI codepage</td>
</tr>
<tr>
<td>encodings.utf_8_sig</td>
<td>UTF-8 codec with BOM signature</td>
</tr>
<tr>
<td>ensurepip</td>
<td>Bootstrapping the &quot;pip&quot; installer into an existing Python installation or virtual environment.</td>
</tr>
<tr>
<td>enum</td>
<td>Implementation of an enumeration class</td>
</tr>
<tr>
<td>Package</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>errno</strong></td>
<td>Standard errno system symbols.</td>
</tr>
<tr>
<td><strong>faulthandler</strong></td>
<td>Dump the Python traceback.</td>
</tr>
<tr>
<td><strong>fcntl</strong> <em>(Unix)</em></td>
<td>The fcntl() and ioctl() system calls.</td>
</tr>
<tr>
<td><strong>filecmp</strong></td>
<td>Compare files efficiently.</td>
</tr>
<tr>
<td><strong>fileinput</strong></td>
<td>Loop over standard input or a list of files.</td>
</tr>
<tr>
<td><strong>fnmatch</strong></td>
<td>Unix shell style filename pattern matching.</td>
</tr>
<tr>
<td><strong>formatter</strong></td>
<td><strong>Deprecated:</strong> Generic output formatter and device interface.</td>
</tr>
<tr>
<td><strong>fractions</strong></td>
<td>Rational numbers.</td>
</tr>
<tr>
<td><strong>ftplib</strong></td>
<td>FTP protocol client (requires sockets).</td>
</tr>
<tr>
<td><strong>functools</strong></td>
<td>Higher-order functions and operations on callable objects.</td>
</tr>
<tr>
<td><strong>gc</strong></td>
<td>Interface to the cycle-detecting garbage collector.</td>
</tr>
<tr>
<td><strong>getopt</strong></td>
<td>Portable parser for command line options; support both short and long option names.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>getpass</td>
<td>Portable reading of passwords and retrieval of the userid.</td>
</tr>
<tr>
<td>gettext</td>
<td>Multilingual internationalization services.</td>
</tr>
<tr>
<td>glob</td>
<td>Unix shell style pathname pattern expansion.</td>
</tr>
<tr>
<td>grp (Unix)</td>
<td>The group database (getgrnam() and friends).</td>
</tr>
<tr>
<td>gzip</td>
<td>Interfaces for gzip compression and decompression using file objects.</td>
</tr>
<tr>
<td>h</td>
<td>Secure hash and message digest algorithms.</td>
</tr>
<tr>
<td>heapq</td>
<td>Heap queue algorithm (a.k.a. priority queue).</td>
</tr>
<tr>
<td>hmac</td>
<td>Keyed-Hashing for Message Authentication (HMAC) implementation</td>
</tr>
<tr>
<td>html</td>
<td>Helpers for manipulating HTML.</td>
</tr>
<tr>
<td>html.entities</td>
<td>Definitions of HTML general entities.</td>
</tr>
<tr>
<td>html.parser</td>
<td>A simple parser that can handle HTML and XHTML.</td>
</tr>
<tr>
<td>http</td>
<td>HTTP status codes and</td>
</tr>
<tr>
<td>Library</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>http.client</code></td>
<td>HTTP and HTTPS protocol client (requires sockets).</td>
</tr>
<tr>
<td><code>http.cookiejar</code></td>
<td>Classes for automatic handling of HTTP cookies.</td>
</tr>
<tr>
<td><code>http.cookies</code></td>
<td>Support for HTTP state management (cookies).</td>
</tr>
<tr>
<td><code>http.server</code></td>
<td>HTTP server and request handlers.</td>
</tr>
<tr>
<td><code>imaplib</code></td>
<td>IMAP4 protocol client (requires sockets).</td>
</tr>
<tr>
<td><code>imghdr</code></td>
<td>Determine the type of image contained in a file or byte stream.</td>
</tr>
<tr>
<td><code>imp</code></td>
<td><strong>Deprecated</strong>: Access the implementation of the import statement.</td>
</tr>
<tr>
<td><code>importlib</code></td>
<td>The implementation of the import machinery.</td>
</tr>
<tr>
<td><code>importlib.abc</code></td>
<td>Abstract base classes related to import</td>
</tr>
<tr>
<td><code>importlib.machinery</code></td>
<td>Importers and path hooks</td>
</tr>
<tr>
<td><code>importlib.resources</code></td>
<td>Package resource reading, opening, and access</td>
</tr>
<tr>
<td><code>importlib.util</code></td>
<td>Utility code for importers</td>
</tr>
<tr>
<td><code>inspect</code></td>
<td>Extract information and</td>
</tr>
<tr>
<td>package</td>
<td>description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>io</td>
<td>Core tools for working with streams.</td>
</tr>
<tr>
<td>ipaddress</td>
<td>IPv4/IPv6 manipulation library.</td>
</tr>
<tr>
<td>itertools</td>
<td>Functions creating iterators for efficient looping.</td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>json</td>
<td>Encode and decode the JSON format.</td>
</tr>
<tr>
<td>json.tool</td>
<td>A command line to validate and pretty-print JSON.</td>
</tr>
<tr>
<td>k</td>
<td></td>
</tr>
<tr>
<td>keyword</td>
<td>Test whether a string is a keyword in Python.</td>
</tr>
<tr>
<td>l</td>
<td></td>
</tr>
<tr>
<td>lib2to3</td>
<td>the 2to3 library</td>
</tr>
<tr>
<td>linecache</td>
<td>This module provides random access to individual lines from text files.</td>
</tr>
<tr>
<td>locale</td>
<td>Internationalization services.</td>
</tr>
<tr>
<td>logging</td>
<td>Flexible event logging system for applications.</td>
</tr>
<tr>
<td>logging.config</td>
<td>Configuration of the logging module.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>logging.handlers</td>
<td>Handlers for the logging module.</td>
</tr>
<tr>
<td>lzma</td>
<td>A Python wrapper for the liblzma compression library.</td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>m pathname</td>
<td>Mac OS 9 path manipulation functions.</td>
</tr>
<tr>
<td>mailbox</td>
<td>Manipulate mailboxes in various formats.</td>
</tr>
<tr>
<td>mailcap</td>
<td>Mailcap file handling.</td>
</tr>
<tr>
<td>marshal</td>
<td>Convert Python objects to streams of bytes and back (with different constraints).</td>
</tr>
<tr>
<td>math</td>
<td>Mathematical functions (sin() etc.).</td>
</tr>
<tr>
<td>mimetypes</td>
<td>Mapping of filename extensions to MIME types.</td>
</tr>
<tr>
<td>mmap</td>
<td>Interface to memory-mapped files for Unix and Windows.</td>
</tr>
<tr>
<td>modulefinder</td>
<td>Find modules used by a script.</td>
</tr>
<tr>
<td>msilib (Windows)</td>
<td>Creation of Microsoft Installer files, and CAB files.</td>
</tr>
<tr>
<td>msvcr (Windows)</td>
<td>Miscellaneous useful routines from the MS VC++ runtime.</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>Process-based parallelism.</td>
</tr>
</tbody>
</table>
multiprocessing.connection: API for dealing with sockets.

multiprocessing.dummy: Dumb wrapper around threading.

multiprocessing.managers: Share data between process with shared objects.

multiprocessing.pool: Create pools of processes.

multiprocessing.sharedctypes: Allocate ctypes objects from shared memory.

<table>
<thead>
<tr>
<th>n</th>
<th>Loading of .netrc files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>netrc</td>
<td>Loading of .netrc files.</td>
</tr>
<tr>
<td>nis (Unix)</td>
<td>Interface to Sun's NIS (Yellow Pages) library.</td>
</tr>
<tr>
<td>nntplib</td>
<td>NNTP protocol client (requires sockets).</td>
</tr>
<tr>
<td>numbers</td>
<td>Numeric abstract base classes (Complex, Real, Integral, etc.).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>o</th>
<th>Functions corresponding to the standard operators.</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator</td>
<td>Functions corresponding to the standard operators.</td>
</tr>
<tr>
<td>Deprecated:</td>
<td>Command-line option parsing library.</td>
</tr>
<tr>
<td>optparse</td>
<td>Command-line option parsing library.</td>
</tr>
<tr>
<td>numbers</td>
<td>Miscellaneous operating system interfaces.</td>
</tr>
<tr>
<td>os</td>
<td>Operations on</td>
</tr>
<tr>
<td>os.path</td>
<td>Operations on</td>
</tr>
</tbody>
</table>
## ossaudiodev *(Linux, FreeBSD)*
Access to OSS-compatible audio devices.

<table>
<thead>
<tr>
<th>p</th>
<th>parser</th>
<th>Access parse trees for Python source code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathlib</td>
<td>Object-oriented filesystem paths</td>
<td></td>
</tr>
<tr>
<td>pdb</td>
<td>The Python debugger for interactive interpreters.</td>
<td></td>
</tr>
<tr>
<td>pickle</td>
<td>Convert Python objects to streams of bytes and back.</td>
<td></td>
</tr>
<tr>
<td>pickletools</td>
<td>Contains extensive comments about the pickle protocols and pickle-machine opcodes, as well as some useful functions.</td>
<td></td>
</tr>
<tr>
<td>pipes <em>(Unix)</em></td>
<td>A Python interface to Unix shell pipelines.</td>
<td></td>
</tr>
<tr>
<td>pkgutil</td>
<td>Utilities for the import system.</td>
<td></td>
</tr>
<tr>
<td>platform</td>
<td>Retrieves as much platform identifying data as possible.</td>
<td></td>
</tr>
<tr>
<td>plistlib</td>
<td>Generate and parse Mac OS X plist files.</td>
<td></td>
</tr>
<tr>
<td>poplib</td>
<td>POP3 protocol client (requires sockets). The most common</td>
<td></td>
</tr>
<tr>
<td><strong>posix</strong> <em>(Unix)</em></td>
<td>POSIX system calls (normally used via module os).</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>pprint</strong></td>
<td>Data pretty printer.</td>
<td></td>
</tr>
<tr>
<td><strong>profile</strong></td>
<td>Python source profiler.</td>
<td></td>
</tr>
<tr>
<td><strong>pstats</strong></td>
<td>Statistics object for use with the profiler.</td>
<td></td>
</tr>
<tr>
<td><strong>pty</strong> <em>(Linux)</em></td>
<td>Pseudo-Terminal Handling for Linux.</td>
<td></td>
</tr>
<tr>
<td><strong>pwd</strong> <em>(Unix)</em></td>
<td>The password database (getpwnam() and friends).</td>
<td></td>
</tr>
<tr>
<td><strong>py_compile</strong></td>
<td>Generate byte-code files from Python source files.</td>
<td></td>
</tr>
<tr>
<td><strong>pyclbr</strong></td>
<td>Supports information extraction for a Python class browser.</td>
<td></td>
</tr>
<tr>
<td><strong>pydoc</strong></td>
<td>Documentation generator and online help system.</td>
<td></td>
</tr>
<tr>
<td><strong>q</strong></td>
<td>A synchronized queue class.</td>
<td></td>
</tr>
<tr>
<td><strong>queue</strong></td>
<td>Encode and decode files using the MIME quoted-printable encoding.</td>
<td></td>
</tr>
<tr>
<td><strong>quopri</strong></td>
<td>Generate pseudo-</td>
<td></td>
</tr>
</tbody>
</table>
random numbers with various common distributions.

re

Regular expression operations.

readline (Unix)

GNU readline support for Python.

reprlib

Alternate repr() implementation with size limits.

resource (Unix)

An interface to provide resource usage information on the current process.

rlcompleter

Python identifier completion, suitable for the GNU readline library.

runpy

Locate and run Python modules without importing them first.

<table>
<thead>
<tr>
<th>s</th>
<th>General purpose event scheduler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched</td>
<td>Generate secure random numbers for managing secrets.</td>
</tr>
<tr>
<td>secrets</td>
<td>Wait for I/O completion on multiple streams.</td>
</tr>
<tr>
<td>select</td>
<td>High-level I/O multiplexing.</td>
</tr>
<tr>
<td>selectors</td>
<td>Python object persistence.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>shlex</td>
<td>Simple lexical analysis for Unix shell-like languages.</td>
</tr>
<tr>
<td>shutil</td>
<td>High-level file operations, including copying.</td>
</tr>
<tr>
<td>signal</td>
<td>Set handlers for asynchronous events.</td>
</tr>
<tr>
<td>site</td>
<td>Module responsible for site-specific configuration.</td>
</tr>
<tr>
<td>smtplib</td>
<td>A SMTP server implementation in Python.</td>
</tr>
<tr>
<td>smtpd</td>
<td>SMTP protocol client (requires sockets).</td>
</tr>
<tr>
<td>sndhdr</td>
<td>Determine type of a sound file.</td>
</tr>
<tr>
<td>socket</td>
<td>Low-level networking interface.</td>
</tr>
<tr>
<td>socketserver</td>
<td>A framework for network servers.</td>
</tr>
<tr>
<td>spwd (Unix)</td>
<td>The shadow password database (getspnam() and friends).</td>
</tr>
<tr>
<td>sqlite3</td>
<td>A DB-API 2.0 implementation using SQLite 3.x.</td>
</tr>
<tr>
<td>ssl</td>
<td>TLS/SSL wrapper for socket objects</td>
</tr>
<tr>
<td>stat</td>
<td>Utilities for interpreting the results of os.stat(), os.lstat() and os.fstat().</td>
</tr>
<tr>
<td>statistics</td>
<td>Mathematical statistics</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>string</strong></td>
<td>Common string operations.</td>
</tr>
<tr>
<td><strong>stringprep</strong></td>
<td>String preparation, as per RFC 3453</td>
</tr>
<tr>
<td><strong>struct</strong></td>
<td>Interpret bytes as packed binary data.</td>
</tr>
<tr>
<td><strong>subprocess</strong></td>
<td>Subprocess management.</td>
</tr>
<tr>
<td><strong>sunau</strong></td>
<td>Provide an interface to the Sun AU sound format.</td>
</tr>
<tr>
<td><strong>symbol</strong></td>
<td>Constants representing internal nodes of the parse tree.</td>
</tr>
<tr>
<td><strong>symtable</strong></td>
<td>Interface to the compiler's internal symbol tables.</td>
</tr>
<tr>
<td><strong>sys</strong></td>
<td>Access system-specific parameters and functions.</td>
</tr>
<tr>
<td><strong>sysconfig</strong></td>
<td>Python's configuration information</td>
</tr>
<tr>
<td><strong>syslog</strong></td>
<td>An interface to the Unix syslog library routines.</td>
</tr>
<tr>
<td><strong>tabnanny</strong></td>
<td>Tool for detecting white space related problems in Python source files in a directory tree.</td>
</tr>
<tr>
<td><strong>tarfile</strong></td>
<td>Read and write tar-format archive files.</td>
</tr>
</tbody>
</table>
**telnetlib**  
Telnet client class.

**tempfile**  
Generate temporary files and directories.

**termios** *(Unix)*  
POSIX style tty control.

**test**  
Regression tests package containing the testing suite for Python.

**test.support**  
Support for Python's regression test suite.

**test.support.script_helper**  
Support for Python's script execution tests.

**textwrap**  
Text wrapping and filling

**threading**  
Thread-based parallelism.

**time**  
Time access and conversions.

**timeit**  
Measure the execution time of small code snippets.

**tkinter**  
Interface to Tcl/Tk for graphical user interfaces

**tkinter.scrolledtext** *(Tk)*  
Text widget with a vertical scroll bar.

**tkinter.tix**  
Tk Extension Widgets for Tkinter

**tkinter.ttk**  
Tk themed widget set

**token**  
Constants representing terminal nodes of the parse tree.

**tokenize**  
Lexical scanner for Python source code.

**trace**  
Trace or track Python
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceback</td>
<td>statement execution. Print or retrieve a stack traceback.</td>
</tr>
<tr>
<td>tracemalloc</td>
<td>Trace memory allocations.</td>
</tr>
<tr>
<td>tty (Unix)</td>
<td>Utility functions that perform common terminal control operations.</td>
</tr>
<tr>
<td>turtle</td>
<td>An educational framework for simple graphics applications.</td>
</tr>
<tr>
<td>turtledemo</td>
<td>A viewer for example turtle scripts.</td>
</tr>
<tr>
<td>types</td>
<td>Names for built-in types.</td>
</tr>
<tr>
<td>typing</td>
<td>Support for type hints (see PEP 484).</td>
</tr>
<tr>
<td>univodedata</td>
<td>Access the Unicode Database.</td>
</tr>
<tr>
<td>unittest</td>
<td>Unit testing framework for Python.</td>
</tr>
<tr>
<td>unittest.mock</td>
<td>Mock object library.</td>
</tr>
<tr>
<td>urllib</td>
<td>Exception classes raised by urllib.request.</td>
</tr>
<tr>
<td>urllib.error</td>
<td>Parse URLs into or assemble them from components.</td>
</tr>
<tr>
<td>urllib.parse</td>
<td>Extensible library for opening URLs.</td>
</tr>
<tr>
<td>Module</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>urllib.response</td>
<td>Response classes used by urllib.</td>
</tr>
<tr>
<td>urllib.robotparser</td>
<td>Load a robots.txt file and answer questions about fetchability of other URLs.</td>
</tr>
<tr>
<td>uu</td>
<td>Encode and decode files in uuencode format.</td>
</tr>
<tr>
<td>uuid</td>
<td>UUID objects (universally unique identifiers) according to RFC 4122</td>
</tr>
<tr>
<td>v</td>
<td>Creation of virtual environments.</td>
</tr>
<tr>
<td>venv</td>
<td></td>
</tr>
<tr>
<td>warnings</td>
<td>Issue warning messages and control their disposition.</td>
</tr>
<tr>
<td>wave</td>
<td>Provide an interface to the WAV sound format.</td>
</tr>
<tr>
<td>weakref</td>
<td>Support for weak references and weak dictionaries.</td>
</tr>
<tr>
<td>webbrowser</td>
<td>Easy-to-use controller for Web browsers.</td>
</tr>
<tr>
<td>winreg (Windows)</td>
<td>Routines and objects for manipulating the Windows registry.</td>
</tr>
<tr>
<td>winsound (Windows)</td>
<td>Access to the sound-playing machinery for</td>
</tr>
</tbody>
</table>
```

wsgiref

```Windows.
WSGI Utilities and Reference Implementation.
```

```

wsgiref.handlers

```WSGI server/gateway base classes.
```

```

wsgiref.headers

```WSGI response header tools.
```

```

wsgiref.simple_server

```A simple WSGI HTTP server.
```

```

wsgiref.util

```WSGI environment utilities.
```

```

wsgiref.validate

```WSGI conformance checker.
```

```

<table>
<thead>
<tr>
<th>x</th>
<th>xdrlib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encoders and decoders for the External Data Representation (XDR).</td>
</tr>
</tbody>
</table>

```

```

xml

Package containing XML processing modules.
```

```

xml.dom

```Document Object Model API for Python.
```

```

xml.dom.minidom

```Minimal Document Object Model (DOM) implementation.
```

```

xml.dom.pulldom

```Support for building partial DOM trees from SAX events.
```

```

xml.etree.ElementTree

```Implementation of the ElementTree API.
```

```

xml.parsers.expat

```An interface to the`
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml.parsers.expat.errors</code></td>
<td>Expat non-validating XML parser.</td>
</tr>
<tr>
<td><code>xml.parsers.expat.model</code></td>
<td>Package containing SAX2 base classes and convenience functions.</td>
</tr>
<tr>
<td><code>xml.sax</code></td>
<td>Base classes for SAX event handlers.</td>
</tr>
<tr>
<td><code>xml.sax.handler</code></td>
<td>Convenience functions and classes for use with SAX.</td>
</tr>
<tr>
<td><code>xml.sax.saxutils</code></td>
<td>Interface which SAX-compliant XML parsers must implement.</td>
</tr>
<tr>
<td><code>xml.sax.xmlreader</code></td>
<td></td>
</tr>
<tr>
<td><code>xmlrpc</code></td>
<td></td>
</tr>
<tr>
<td><code>xmlrpc.client</code></td>
<td>XML-RPC client access.</td>
</tr>
<tr>
<td><code>xmlrpc.server</code></td>
<td>Basic XML-RPC server implementations.</td>
</tr>
<tr>
<td><code>zipapp</code></td>
<td>Manage executable Python zip archives</td>
</tr>
<tr>
<td><code>zipfile</code></td>
<td>Read and write ZIP-format archive files.</td>
</tr>
<tr>
<td><code>zipimport</code></td>
<td>Support for importing Python modules from ZIP archives.</td>
</tr>
<tr>
<td><code>zlib</code></td>
<td>Low-level interface to compression and decompression routines compatible with gzip.</td>
</tr>
</tbody>
</table>
Welcome! This is the documentation for Python 3.7.3.

Parts of the documentation:

What's new in Python 3.7? or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Indices and tables:
What’s New In Python 3.7

This article explains the new features in Python 3.7, compared to 3.6. Python 3.7 was released on June 27, 2018. For full details, see the changelog.
Summary – Release Highlights

New syntax features:

- **PEP 563**, postponed evaluation of type annotations.

Backwards incompatible syntax changes:

- `async` and `await` are now reserved keywords.

New library modules:

- **contextvars**: PEP 567 – Context Variables
- **dataclasses**: PEP 557 – Data Classes
- **importlib.resources**

New built-in features:

- **PEP 553**, the new `breakpoint()` function.

Python data model improvements:

- **PEP 562**, customization of access to module attributes.
- **PEP 560**, core support for typing module and generic types.
- the insertion-order preservation nature of `dict` objects has been declared to be an official part of the Python language spec.

Significant improvements in the standard library:

- The `asyncio` module has received new features, significant usability and performance improvements.
- The `time` module gained support for functions with nanosecond resolution.

CPython implementation improvements:
• Avoiding the use of ASCII as a default text encoding:
  ○ PEP 538, legacy C locale coercion
  ○ PEP 540, forced UTF-8 runtime mode
• PEP 552, deterministic .pycs
• the new development runtime mode
• PEP 565, improved DeprecationWarning handling

C API improvements:

• PEP 539, new C API for thread-local storage

Documentation improvements:

• PEP 545, Python documentation translations

This release features notable performance improvements in many areas. The Optimizations section lists them in detail.

For a list of changes that may affect compatibility with previous Python releases please refer to the Porting to Python 3.7 section.
New Features

PEP 563: Postponed Evaluation of Annotations

The advent of type hints in Python uncovered two glaring usability issues with the functionality of annotations added in PEP 3107 and refined further in PEP 526:

- annotations could only use names which were already available in the current scope, in other words they didn’t support forward references of any kind; and
- annotating source code had adverse effects on startup time of Python programs.

Both of these issues are fixed by postponing the evaluation of annotations. Instead of compiling code which executes expressions in annotations at their definition time, the compiler stores the annotation in a string form equivalent to the AST of the expression in question. If needed, annotations can be resolved at runtime using typing.get_type_hints(). In the common case where this is not required, the annotations are cheaper to store (since short strings are interned by the interpreter) and make startup time faster.

Usability-wise, annotations now support forward references, making the following syntax valid:

```python
class C:
    @classmethod
    def from_string(cls, source: str) -> C:
        ...

    def validate_b(self, obj: B) -> bool:
        ...
```
class B:
...

Since this change breaks compatibility, the new behavior needs to be enabled on a per-module basis in Python 3.7 using a `__future__` import:

```python
from __future__ import annotations
```

It will become the default in Python 4.0.

**See also:**

**PEP 563** – Postponed evaluation of annotations
PEP written and implemented by Łukasz Langa.

**PEP 538: Legacy C Locale Coercion**

An ongoing challenge within the Python 3 series has been determining a sensible default strategy for handling the “7-bit ASCII” text encoding assumption currently implied by the use of the default C or POSIX locale on non-Windows platforms.

**PEP 538** updates the default interpreter command line interface to automatically coerce that locale to an available UTF-8 based locale as described in the documentation of the new `PYTHONCOERCECLOCALE` environment variable. Automatically setting `LC_CTYPE` this way means that both the core interpreter and locale-aware C extensions (such as `readline`) will assume the use of UTF-8 as the default text encoding, rather than ASCII.

The platform support definition in **PEP 11** has also been updated to
limit full text handling support to suitably configured non-ASCII based locales.

As part of this change, the default error handler for stdin and stdout is now surrogateescape (rather than strict) when using any of the defined coercion target locales (currently C.UTF-8, C.utf8, and UTF-8). The default error handler for stderr continues to be backslashreplace, regardless of locale.

Locale coercion is silent by default, but to assist in debugging potentially locale related integration problems, explicit warnings (emitted directly on stderr) can be requested by setting PYTHONCOERCECLOCALE=warn. This setting will also cause the Python runtime to emit a warning if the legacy C locale remains active when the core interpreter is initialized.

While PEP 538's locale coercion has the benefit of also affecting extension modules (such as GNU readline), as well as child processes (including those running non-Python applications and older versions of Python), it has the downside of requiring that a suitable target locale be present on the running system. To better handle the case where no suitable target locale is available (as occurs on RHEL/CentOS 7, for example), Python 3.7 also implements PEP 540: Forced UTF-8 Runtime Mode.

See also:

PEP 538 – Coercing the legacy C locale to a UTF-8 based locale

PEP written and implemented by Nick Coghlan.

PEP 540: Forced UTF-8 Runtime Mode
The new `-X utf8` command line option and `PYTHONUTF8` environment variable can be used to enable the CPython UTF-8 mode.

When in UTF-8 mode, CPython ignores the locale settings, and uses the UTF-8 encoding by default. The error handlers for `sys.stdin` and `sys.stdout` streams are set to surrogateescape.

The forced UTF-8 mode can be used to change the text handling behavior in an embedded Python interpreter without changing the locale settings of an embedding application.

While PEP 540’s UTF-8 mode has the benefit of working regardless of which locales are available on the running system, it has the downside of having no effect on extension modules (such as GNU readline), child processes running non-Python applications, and child processes running older versions of Python. To reduce the risk of corrupting text data when communicating with such components, Python 3.7 also implements PEP 540: Forced UTF-8 Runtime Mode).

The UTF-8 mode is enabled by default when the locale is C or POSIX, and the PEP 538 locale coercion feature fails to change it to a UTF-8 based alternative (whether that failure is due to `PYTHONCOERCECLOCALE=0` being set, `LC_ALL` being set, or the lack of a suitable target locale).

See also:

**PEP 540** – Add a new UTF-8 mode
  
  PEP written and implemented by Victor Stinner

PEP 553: Built-in breakpoint()
Python 3.7 includes the new built-in `breakpoint()` function as an easy and consistent way to enter the Python debugger.

Built-in `breakpoint()` calls `sys.breakpointhook()`. By default, the latter imports `pdb` and then calls `pdb.set_trace()`, but by binding `sys.breakpointhook()` to the function of your choosing, `breakpoint()` can enter any debugger. Additionally, the environment variable `PYTHONBREAKPOINT` can be set to the callable of your debugger of choice. Set `PYTHONBREAKPOINT=0` to completely disable built-in `breakpoint()`.

**See also:**

**PEP 553** – Built-in `breakpoint()`

PEP written and implemented by Barry Warsaw

### PEP 539: New C API for Thread-Local Storage

While Python provides a C API for thread-local storage support; the existing **Thread Local Storage (TLS) API** has used `int` to represent TLS keys across all platforms. This has not generally been a problem for officially-support platforms, but that is neither POSIX-compliant, nor portable in any practical sense.

**PEP 539** changes this by providing a new **Thread Specific Storage (TSS) API** to CPython which supersedes use of the existing TLS API within the CPython interpreter, while deprecating the existing API. The TSS API uses a new type `Py_tss_t` instead of `int` to represent TSS keys—an opaque type the definition of which may depend on the underlying TLS implementation. Therefore, this will allow to build CPython on platforms where the native TLS key is defined in a way
that cannot be safely cast to `int`.

Note that on platforms where the native TLS key is defined in a way that cannot be safely cast to `int`, all functions of the existing TLS API will be no-op and immediately return failure. This indicates clearly that the old API is not supported on platforms where it cannot be used reliably, and that no effort will be made to add such support.

**See also:**

**PEP 539** – A New C-API for Thread-Local Storage in CPython  
PEP written by Erik M. Bray; implementation by Masayuki Yamamoto.

**PEP 562: Customization of Access to Module Attributes**

Python 3.7 allows defining `__getattr__()` on modules and will call it whenever a module attribute is otherwise not found. Defining `__dir__()` on modules is now also allowed.

A typical example of where this may be useful is module attribute deprecation and lazy loading.

**See also:**

**PEP 562** – Module `__getattr__` and `__dir__`  
PEP written and implemented by Ivan Levkivskyi

**PEP 564: New Time Functions With Nanosecond Resolution**
The resolution of clocks in modern systems can exceed the limited precision of a floating point number returned by the `time.time()` function and its variants. To avoid loss of precision, **PEP 564** adds six new “nanosecond” variants of the existing timer functions to the `time` module:

- `time.clock_gettime_ns()`
- `time.clock_settime_ns()`
- `time.monotonic_ns()`
- `time.perf_counter_ns()`
- `time.process_time_ns()`
- `time.time_ns()`

The new functions return the number of nanoseconds as an integer value.

**Measurements** show that on Linux and Windows the resolution of `time.time_ns()` is approximately 3 times better than that of `time.time()`.

---

**See also:**

**PEP 564** – Add new time functions with nanosecond resolution

PEP written and implemented by Victor Stinner

---

**PEP 565: Show DeprecationWarning in __main__**

The default handling of `DeprecationWarning` has been changed such that these warnings are once more shown by default, but only when the code triggering them is running directly in the `__main__` module. As a result, developers of single file scripts and those using
Python interactively should once again start seeing deprecation warnings for the APIs they use, but deprecation warnings triggered by imported application, library and framework modules will continue to be hidden by default.

As a result of this change, the standard library now allows developers to choose between three different deprecation warning behaviours:

- **FutureWarning**: always displayed by default, recommended for warnings intended to be seen by application end users (e.g. for deprecated application configuration settings).
- **DeprecationWarning**: displayed by default only in `__main__` and when running tests, recommended for warnings intended to be seen by other Python developers where a version upgrade may result in changed behaviour or an error.
- **PendingDeprecationWarning**: displayed by default only when running tests, intended for cases where a future version upgrade will change the warning category to `DeprecationWarning` or `FutureWarning`.

Previously, both `DeprecationWarning` and `PendingDeprecationWarning` were only visible when running tests, which meant that developers primarily writing single file scripts or using Python interactively could be surprised by breaking changes in the APIs they used.

**See also:**

**PEP 565 – Show DeprecationWarning in `__main__`**
PEP written and implemented by Nick Coghlan

**PEP 560: Core Support for typing module and**
Generic Types

Initially PEP 484 was designed in such way that it would not introduce any changes to the core CPython interpreter. Now type hints and the `typing` module are extensively used by the community, so this restriction is removed. The PEP introduces two special methods `__class_getitem__()` and `__mro_entries__`, these methods are now used by most classes and special constructs in `typing`. As a result, the speed of various operations with types increased up to 7 times, the generic types can be used without metaclass conflicts, and several long standing bugs in `typing` module are fixed.

See also:

**PEP 560** – Core support for typing module and generic types

PEP written and implemented by Ivan Levkivskyi

PEP 552: Hash-based .pyc Files

Python has traditionally checked the up-to-dateness of bytecode cache files (i.e., `.pyc` files) by comparing the source metadata (last-modified timestamp and size) with source metadata saved in the cache file header when it was generated. While effective, this invalidation method has its drawbacks. When filesystem timestamps are too coarse, Python can miss source updates, leading to user confusion. Additionally, having a timestamp in the cache file is problematic for build reproducibility and content-based build systems.

**PEP 552** extends the pyc format to allow the hash of the source file to be used for invalidation instead of the source timestamp. Such `.pyc` files are called “hash-based”. By default, Python still uses timestamp-
based invalidation and does not generate hash-based .pyc files at runtime. Hash-based .pyc files may be generated with py_compile or compileall.

Hash-based .pyc files come in two variants: checked and unchecked. Python validates checked hash-based .pyc files against the corresponding source files at runtime but doesn’t do so for unchecked hash-based pycs. Unchecked hash-based .pyc files are a useful performance optimization for environments where a system external to Python (e.g., the build system) is responsible for keeping .pyc files up-to-date.

See Cached bytecode invalidation for more information.

See also:

**PEP 552 – Deterministic pycs**
PEP written and implemented by Benjamin Peterson

PEP 545: Python Documentation Translations

**PEP 545** describes the process of creating and maintaining Python documentation translations.

Three new translations have been added:

- Japanese: https://docs.python.org/ja/
- French: https://docs.python.org/fr/
- Korean: https://docs.python.org/ko/

See also:

**PEP 545 – Python Documentation Translations**
Development Runtime Mode: -X dev

The new `-X dev` command line option or the new `PYTHONDEVMODE` environment variable can be used to enable CPython's *development mode*. When in development mode, CPython performs additional runtime checks that are too expensive to be enabled by default. See `-X dev` documentation for the full description of the effects of this mode.
Other Language Changes

- More than 255 arguments can now be passed to a function, and a function can now have more than 255 parameters. (Contributed by Serhiy Storchaka in bpo-12844 and bpo-18896.)
- `bytes.fromhex()` and `bytearray.fromhex()` now ignore all ASCII whitespace, not only spaces. (Contributed by Robert Xiao in bpo-28927.)
- `str`, `bytes`, and `bytearray` gained support for the new `isascii()` method, which can be used to test if a string or bytes contain only the ASCII characters. (Contributed by INADA Naoki in bpo-32677.)
- `ImportError` now displays module name and module `__file__` path when `from ... import ...` fails. (Contributed by Matthias Bussonnier in bpo-29546.)
- Circular imports involving absolute imports with binding a submodule to a name are now supported. (Contributed by Serhiy Storchaka in bpo-30024.)
- `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(self), '')`. (Contributed by Serhiy Storchaka in bpo-28974.)
- In order to better support dynamic creation of stack traces, `types.TracebackType` can now be instantiated from Python code, and the `tb_next` attribute on `tracebacks` is now writable. (Contributed by Nathaniel J. Smith in bpo-30579.)
- When using the `-m` switch, `sys.path[0]` is now eagerly expanded to the full starting directory path, rather than being left as the empty directory (which allows imports from the current working directory at the time when an import occurs) (Contributed by Nick Coghlan in bpo-33053.)
- The new `-X importtime` option or the `PYTHONPROFILEIMPORTTIME` environment variable can be used
to show the timing of each module import. (Contributed by Victor Stinner in bpo-31415.)
New Modules

contextvars

The new `contextvars` module and a set of new C APIs introduce support for context variables. Context variables are conceptually similar to thread-local variables. Unlike TLS, context variables support asynchronous code correctly.

The `asyncio` and `decimal` modules have been updated to use and support context variables out of the box. Particularly the active decimal context is now stored in a context variable, which allows decimal operations to work with the correct context in asynchronous code.

See also:

**PEP 567 – Context Variables**

PEP written and implemented by Yury Selivanov

dataclasses

The new `dataclass()` decorator provides a way to declare data classes. A data class describes its attributes using class variable annotations. Its constructor and other magic methods, such as `__repr__()`, `__eq__()`, and `__hash__()` are generated automatically.

Example:

```python
@dataclass
class Point:
    x: float
```
y: float
z: float = 0.0

p = Point(1.5, 2.5)
print(p)  # produces "Point(x=1.5, y=2.5, z=0.0)"

See also:

PEP 557 – Data Classes
PEP written and implemented by Eric V. Smith

importlib.resources

The new importlib.resources module provides several new APIs and one new ABC for access to, opening, and reading resources inside packages. Resources are roughly similar to files inside packages, but they needn’t be actual files on the physical file system. Module loaders can provide a get_resource_reader() function which returns a importlib.abc.ResourceReader instance to support this new API. Built-in file path loaders and zip file loaders both support this.

Contributed by Barry Warsaw and Brett Cannon in bpo-32248.

See also: importlib_resources – a PyPI backport for earlier Python versions.
Improved Modules

argparse

The new `ArgumentParser.parse_intermixed_args()` method allows intermixing options and positional arguments. (Contributed by paul.j3 in bpo-14191.)

asyncio

The `asyncio` module has received many new features, usability and performance improvements. Notable changes include:

- The new `provisional asyncio.run()` function can be used to run a coroutine from synchronous code by automatically creating and destroying the event loop. (Contributed by Yury Selivanov in bpo-32314.)

- `asyncio` gained support for `contextvars.loop.call_soon()`, `loop.call_soon_threadsafe()`, `loop.call_later()`, `loop.call_at()`, and `Future.add_done_callback()` have a new optional keyword-only `context` parameter. `Tasks` now track their context automatically. See PEP 567 for more details. (Contributed by Yury Selivanov in bpo-32436.)

- The new `asyncio.create_task()` function has been added as a shortcut to `asyncio.get_event_loop().create_task()`. (Contributed by Andrew Svetlov in bpo-32311.)

- The new `loop.start_tls()` method can be used to upgrade
an existing connection to TLS. (Contributed by Yury Selivanov in bpo-23749.)

- The new `loop.sock_recv_into()` method allows reading data from a socket directly into a provided buffer making it possible to reduce data copies. (Contributed by Antoine Pitrou in bpo-31819.)

- The new `asyncio.current_task()` function returns the currently running Task instance, and the new `asyncio.all_tasks()` function returns a set of all existing Task instances in a given loop. The `Task.current_task()` and `Task.all_tasks()` methods have been deprecated. (Contributed by Andrew Svetlov in bpo-32250.)

- The new `provisional BufferedProtocol` class allows implementing streaming protocols with manual control over the receive buffer. (Contributed by Yury Selivanov in bpo-32251.)

- The new `asyncio.get_running_loop()` function returns the currently running loop, and raises a `RuntimeError` if no loop is running. This is in contrast with `asyncio.get_event_loop()`, which will create a new event loop if none is running. (Contributed by Yury Selivanov in bpo-32269.)

- The new `StreamWriter.wait_closed()` coroutine method allows waiting until the stream writer is closed. The new `StreamWriter.is_closing()` method can be used to determine if the writer is closing. (Contributed by Andrew Svetlov in bpo-32391.)

- The new `loop.sock_sendfile()` coroutine method allows
sending files using `os.sendfile` when possible. (Contributed by Andrew Svetlov in bpo-32410.)

- The new `Future.get_loop()` and `Task.get_loop()` methods return the instance of the loop on which a task or a future were created. `Server.get_loop()` allows doing the same for `asyncio.Server` objects. (Contributed by Yury Selivanov in bpo-32415 and Srinivas Reddy Thatiparthy in bpo-32418.)

- It is now possible to control how instances of `asyncio.Server` begin serving. Previously, the server would start serving immediately when created. The new `start_serving` keyword argument to `loop.create_server()` and `loop.create_unix_server()`, as well as `Server.start_serving()`, and `Server.serve_forever()` can be used to decouple server instantiation and serving. The new `Server.is_serving()` method returns `True` if the server is serving. `Server` objects are now asynchronous context managers:

```python
srv = await loop.create_server(...)  
async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

(Contributed by Yury Selivanov in bpo-32662.)

- Callback objects returned by `loop.call_later()` gained the new `when()` method which returns an absolute scheduled
callback timestamp. (Contributed by Andrew Svetlov in bpo-32741.)

- The `loop.create_datagram_endpoint()` method gained support for Unix sockets. (Contributed by Quentin Dawans in bpo-31245.)

- The `asyncio.open_connection()`, `asyncio.start_server()`, `loop.create_connection()`, `loop.create_server()`, `loop.create_accepted_socket()` methods and their corresponding UNIX socket variants now accept the `ssl_handshake_timeout` keyword argument. (Contributed by Neil Aspinall in bpo-29970.)

- The new `Handle.cancelled()` method returns `True` if the callback was cancelled. (Contributed by Marat Sharafutdinov in bpo-31943.)

- The asyncio source has been converted to use the `async/await` syntax. (Contributed by Andrew Svetlov in bpo-32193.)

- The new `ReadTransport.is_reading()` method can be used to determine the reading state of the transport. Additionally, calls to `ReadTransport.resume_reading()` and `ReadTransport.pause_reading()` are now idempotent. (Contributed by Yury Selivanov in bpo-32356.)

- Loop methods which accept socket paths now support passing path-like objects. (Contributed by Yury Selivanov in bpo-32066.)

- In `asyncio` TCP sockets on Linux are now created with `TCP_NODELAY` flag set by default. (Contributed by Yury Selivanov
and Victor Stinner in `bpo-27456`.

- Exceptions occurring in cancelled tasks are no longer logged. (Contributed by Yury Selivanov in `bpo-30508`.)

- New `WindowsSelectorEventLoopPolicy` and `WindowsProactorEventLoopPolicy` classes. (Contributed by Yury Selivanov in `bpo-33792`.)

Several `asyncio` APIs have been *deprecated*.

**binascii**

The `b2a_uu()` function now accepts an optional *backtick* keyword argument. When it’s true, zeros are represented by `'` instead of spaces. (Contributed by Xiang Zhang in `bpo-30103`.)

**calendar**

The `HTMLCalendar` class has new class attributes which ease the customization of CSS classes in the produced HTML calendar. (Contributed by Oz Tiram in `bpo-30095`.)

**collections**

`collections.namedtuple()` now supports default values. (Contributed by Raymond Hettinger in `bpo-32320`.)

**compileall**

`compileall.compile_dir()` learned the new *invalidation_mode*
parameter, which can be used to enable hash-based .pyc invalidation. The invalidation mode can also be specified on the command line using the new --invalidation-mode argument. (Contributed by Benjamin Peterson in bpo-31650.)

**concurrent.futures**

*ProcessPoolExecutor* and *ThreadPoolExecutor* now support the new *initializer* and *initargs* constructor arguments. (Contributed by Antoine Pitrou in bpo-21423.)

The *ProcessPoolExecutor* can now take the multiprocessing context via the new *mp_context* argument. (Contributed by Thomas Moreau in bpo-31540.)

**contextlib**

The new *nullcontext()* is a simpler and faster no-op context manager than *ExitStack*. (Contributed by Jesse-Bakker in bpo-10049.)

The new *asynccontextmanager()* , *AbstractAsyncContextManager*, and *AsyncExitStack* have been added to complement their synchronous counterparts. (Contributed by Jelle Zijlstra in bpo-29679 and bpo-30241, and by Alexander Mohr and Ilya Kulakov in bpo-29302.)

**cProfile**

The *cProfile* command line now accepts -m module_name as an alternative to script path. (Contributed by Sanyam Khurana in bpo-
The *crypt* module now supports the Blowfish hashing method. (Contributed by Serhiy Storchaka in bpo-31664.)

The *mksalt()* function now allows specifying the number of rounds for hashing. (Contributed by Serhiy Storchaka in bpo-31702.)

The *datetime* module

The new *datetime.fromisoformat()* method constructs a *datetime* object from a string in one of the formats output by *datetime.isoformat()*.

The *tzinfo* class now supports sub-minute offsets. (Contributed by Alexander Belopolsky in bpo-5288.)

The *dbm* module

*dbm.dumb* now supports reading read-only files and no longer writes the index file when it is not changed.

The *decimal* module now uses *context variables* to store the decimal context. (Contributed by Yury Selivanov in bpo-32630.)
The `dis()` function is now able to disassemble nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). The maximum depth of disassembly recursion is controlled by the new `depth` parameter. (Contributed by Serhiy Storchaka in `bpo-11822`.)

**distutils**

`README.rst` is now included in the list of distutils standard READMEs and therefore included in source distributions. (Contributed by Ryan Gonzalez in `bpo-11913`.)

**enum**

The `Enum` learned the new `__ignore__` class property, which allows listing the names of properties which should not become enum members. (Contributed by Ethan Furman in `bpo-31801`.)

In Python 3.8, attempting to check for non-Enum objects in Enum classes will raise a `TypeError` (e.g. `1 in Color`); similarly, attempting to check for non-Flag objects in a Flag member will raise `TypeError` (e.g. `1 in Perm.RW`); currently, both operations return `False` instead and are deprecated. (Contributed by Ethan Furman in `bpo-33217`.)

**functools**

`functools.singledispatch()` now supports registering implementations using type annotations. (Contributed by Łukasz Langa in `bpo-32227`.)
The new `gc.freeze()` function allows freezing all objects tracked by the garbage collector and excluding them from future collections. This can be used before a POSIX `fork()` call to make the GC copy-on-write friendly or to speed up collection. The new `gc.unfreeze()` function reverses this operation. Additionally, `gc.get_freeze_count()` can be used to obtain the number of frozen objects. (Contributed by Li Zekun in bpo-31558.)

The `hmac` module now has an optimized one-shot `digest()` function, which is up to three times faster than `HMAC()`. (Contributed by Christian Heimes in bpo-32433.)

`http.client` HTTPConnection and HTTPSConnection now support the new `blocksize` argument for improved upload throughput. (Contributed by Nir Soffer in bpo-31945.)

`http.server` SimpleHTTPRequestHandler now supports the HTTP `If-Modified-Since` header. The server returns the 304 response status if the target file was not modified after the time specified in the header. (Contributed by Pierre Quentel in bpo-29654.)

SimpleHTTPRequestHandler accepts the new `directory` argument,
in addition to the new `--directory` command line argument. With this parameter, the server serves the specified directory, by default it uses the current working directory. (Contributed by Stéphane Wirtel and Julien Palard in bpo-28707.)

The new `ThreadingHTTPServer` class uses threads to handle requests using ThreadingMixin. It is used when `http.server` is run with `--m`. (Contributed by Julien Palard in bpo-31639.)

**idlelib and IDLE**

Multiple fixes for autocompletion. (Contributed by Louie Lu in bpo-15786.)

Module Browser (on the File menu, formerly called Class Browser), now displays nested functions and classes in addition to top-level functions and classes. (Contributed by Guilherme Polo, Cheryl Sabella, and Terry Jan Reedy in bpo-1612262.)

The Settings dialog (Options, Configure IDLE) has been partly rewritten to improve both appearance and function. (Contributed by Cheryl Sabella and Terry Jan Reedy in multiple issues.)

The font sample now includes a selection of non-Latin characters so that users can better see the effect of selecting a particular font. (Contributed by Terry Jan Reedy in bpo-13802.) The sample can be edited to include other characters. (Contributed by Serhiy Storchaka in bpo-31860.)

The IDLE features formerly implemented as extensions have been reimplemented as normal features. Their settings have been moved from the Extensions tab to other dialog tabs. (Contributed by Charles Wohlganger and Terry Jan Reedy in bpo-27099.)
Editor code context option revised. Box displays all context lines up to maxlines. Clicking on a context line jumps the editor to that line. Context colors for custom themes is added to Highlights tab of Settings dialog. (Contributed by Cheryl Sabella and Terry Jan Reedy in bpo-33642, bpo-33768, and bpo-33679.)

On Windows, a new API call tells Windows that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect. (Contributed by Terry Jan Reedy in bpo-33656.)

New in 3.7.1:

Output over N lines (50 by default) is squeezed down to a button. N can be changed in the PyShell section of the General page of the Settings dialog. Fewer, but possibly extra long, lines can be squeezed by right clicking on the output. Squeezed output can be expanded in place by double-clicking the button or into the clipboard or a separate window by right-clicking the button. (Contributed by Tal Einat in bpo-1529353.)

The changes above have been backported to 3.6 maintenance releases.

**importlib**

The `importlib.abc.ResourceReader` ABC was introduced to support the loading of resources from packages. See also `importlib.resources`. (Contributed by Barry Warsaw, Brett Cannon in bpo-32248.)

`importlib.reload()` now raises `ModuleNotFoundError` if the
module lacks a spec. (Contributed by Garvit Khatri in bpo-29851.)

importlib.find_spec() now raises ModuleNotFoundError instead of AttributeError if the specified parent module is not a package (i.e. lacks a __path__ attribute). (Contributed by Milan Oberkirch in bpo-30436.)

The new importlib.source_hash() can be used to compute the hash of the passed source. A hash-based .pyc file embeds the value returned by this function.

io

The new TextIOWrapper.reconfigure() method can be used to reconfigure the text stream with the new settings. (Contributed by Antoine Pitrou in bpo-30526 and INADA Naoki in bpo-15216.)

ipaddress

The new subnet_of() and supernet_of() methods of ipaddress.IPv6Network and ipaddress.IPv4Network can be used for network containment tests. (Contributed by Michel Albert and Cheryl Sabella in bpo-20825.)

itertools

itertools.islice() now accepts integer-like objects as start, stop, and slice arguments. (Contributed by Will Roberts in bpo-30537.)

locale
The new *monetary* argument to `locale.format_string()` can be used to make the conversion use monetary thousands separators and grouping strings. (Contributed by Garvit in bpo-10379.)

The `locale.getpreferredencoding()` function now always returns 'UTF-8' on Android or when in the *forced UTF-8 mode*.

**logging**

*Logger* instances can now be pickled. (Contributed by Vinay Sajip in bpo-30520.)

The new `StreamHandler.setStream()` method can be used to replace the logger stream after handler creation. (Contributed by Vinay Sajip in bpo-30522.)

It is now possible to specify keyword arguments to handler constructors in configuration passed to `logging.config.fileConfig()`. (Contributed by Preston Landers in bpo-31080.)

**math**

The new `math.remainder()` function implements the IEEE 754-style remainder operation. (Contributed by Mark Dickinson in bpo-29962.)

**mimetypes**

The MIME type of .bmp has been changed from 'image/x-ms-bmp' to 'image/bmp'. (Contributed by Nitish Chandra in bpo-22589.)
msilib

The new `Database.Close()` method can be used to close the MSI database. (Contributed by Berker Peksag in bpo-20486.)

multiprocessing

The new `Process.close()` method explicitly closes the process object and releases all resources associated with it. `ValueError` is raised if the underlying process is still running. (Contributed by Antoine Pitrou in bpo-30596.)

The new `Process.kill()` method can be used to terminate the process using the SIGKILL signal on Unix. (Contributed by Vitor Pereira in bpo-30794.)

Non-daemonic threads created by `Process` are now joined on process exit. (Contributed by Antoine Pitrou in bpo-18966.)

OS

`os.fwalk()` now accepts the `path` argument as `bytes`. (Contributed by Serhiy Storchaka in bpo-28682.)

`os.scandir()` gained support for file descriptors. (Contributed by Serhiy Storchaka in bpo-25996.)

The new `register_at_fork()` function allows registering Python callbacks to be executed at process fork. (Contributed by Antoine Pitrou in bpo-16500.)
Added `os.preadv()` (combine the functionality of `os.readv()` and `os.pread()`) and `os.pwritev()` functions (combine the functionality of `os.writev()` and `os.pwrite()`). (Contributed by Pablo Galindo in bpo-31368.)

The mode argument of `os.makedirs()` no longer affects the file permission bits of newly-created intermediate-level directories. (Contributed by Serhiy Storchaka in bpo-19930.)

`os.dup2()` now returns the new file descriptor. Previously, `None` was always returned. (Contributed by Benjamin Peterson in bpo-32441.)

The structure returned by `os.stat()` now contains the `st_fstype` attribute on Solaris and its derivatives. (Contributed by Jesús Cea Avión in bpo-32659.)

**pathlib**

The new `Path.is_mount()` method is now available on POSIX systems and can be used to determine whether a path is a mount point. (Contributed by Cooper Ry Lees in bpo-30897.)

**pdb**

`pdb.set_trace()` now takes an optional `header` keyword-only argument. If given, it is printed to the console just before debugging begins. (Contributed by Barry Warsaw in bpo-31389.)

`pdb` command line now accepts `-m module_name` as an alternative to script file. (Contributed by Mario Corchero in bpo-32206.)

**py_compile**
py_compile.compile() – and by extension, compileall – now respects the SOURCE_DATE_EPOCH environment variable by unconditionally creating .pyc files for hash-based validation. This allows for guaranteeing reproducible builds of .pyc files when they are created eagerly. (Contributed by Bernhard M. Wiedemann in bpo-29708.)

pydoc

The pydoc server can now bind to an arbitrary hostname specified by the new -n command-line argument. (Contributed by Feanil Patel in bpo-31128.)

queue

The new SimpleQueue class is an unbounded FIFO queue. (Contributed by Antoine Pitrou in bpo-14976.)

re

The flags re.ASCII, re.LOCALE and re.UNICODE can be set within the scope of a group. (Contributed by Serhiy Storchaka in bpo-31690.)

re.split() now supports splitting on a pattern like r'\b', '^$' or (?=-) that matches an empty string. (Contributed by Serhiy Storchaka in bpo-25054.)

Regular expressions compiled with the re.LOCALE flag no longer depend on the locale at compile time. Locale settings are applied only
when the compiled regular expression is used. (Contributed by Serhiy Storchaka in bpo-30215.)

**FutureWarning** is now emitted if a regular expression contains character set constructs that will change semantically in the future, such as nested sets and set operations. (Contributed by Serhiy Storchaka in bpo-30349.)

Compiled regular expression and match objects can now be copied using `copy.copy()` and `copy.deepcopy()`. (Contributed by Serhiy Storchaka in bpo-10076.)

**signal**

The new `warn_on_full_buffer` argument to the `signal.set_wakeup_fd()` function makes it possible to specify whether Python prints a warning on stderr when the wakeup buffer overflows. (Contributed by Nathaniel J. Smith in bpo-30050.)

**socket**

The new `socket.getblocking()` method returns `True` if the socket is in blocking mode and `False` otherwise. (Contributed by Yury Selivanov in bpo-32373.)

The new `socket.close()` function closes the passed socket file descriptor. This function should be used instead of `os.close()` for better compatibility across platforms. (Contributed by Christian Heimes in bpo-32454.)

The `socket` module now exposes the `socket.TCP_CONGESTION` (Linux 2.6.13), `socket.TCP_USER_TIMEOUT` (Linux 2.6.37), and
socket.TCP_NOTSENT_LOWAT (Linux 3.12) constants. (Contributed by Omar Sandoval in bpo-26273 and Nathaniel J. Smith in bpo-29728.)

Support for socket.AF_VSOCK sockets has been added to allow communication between virtual machines and their hosts. (Contributed by Cathy Avery in bpo-27584.)

Sockets now auto-detect family, type and protocol from file descriptor by default. (Contributed by Christian Heimes in bpo-28134.)

**socketserver**

socketserver.ThreadingMixIn.server_close() now waits until all non-daemon threads complete.
socketserver.ForkingMixIn.server_close() now waits until all child processes complete.

Add a new socketserver.ForkingMixIn.block_on_close class attribute to socketserver.ForkingMixIn and socketserver.ThreadingMixIn classes. Set the class attribute to False to get the pre-3.7 behaviour.

**sqlite3**

sqlite3.Connection now exposes the backup() method when the underlying SQLite library is at version 3.6.11 or higher. (Contributed by Lele Gaifax in bpo-27645.)

The database argument of sqlite3.connect() now accepts any path-like object, instead of just a string. (Contributed by Anders Lorentsen in bpo-31843.)
The ssl module now uses OpenSSL’s builtin API instead of match_hostname() to check a host name or an IP address. Values are validated during TLS handshake. Any certificate validation error including failing the host name check now raises SSLCertVerificationError and aborts the handshake with a proper TLS Alert message. The new exception contains additional information. Host name validation can be customized with SSLContext.host_flags. (Contributed by Christian Heimes in bpo-31399.)

**Note:** The improved host name check requires a libssl implementation compatible with OpenSSL 1.0.2 or 1.1. Consequently, OpenSSL 0.9.8 and 1.0.1 are no longer supported (see Platform Support Removals for more details). The ssl module is mostly compatible with LibreSSL 2.7.2 and newer.

The ssl module no longer sends IP addresses in SNI TLS extension. (Contributed by Christian Heimes in bpo-32185.)

match_hostname() no longer supports partial wildcards like www*.example.org. SSLContext.host_flags has partial wildcard matching disabled by default. (Contributed by Mandeep Singh in bpo-23033 and Christian Heimes in bpo-31399.)

The default cipher suite selection of the ssl module now uses a blacklist approach rather than a hard-coded whitelist. Python no longer re-enables ciphers that have been blocked by OpenSSL security updates. Default cipher suite selection can be configured at compile time. (Contributed by Christian Heimes in bpo-31429.)
Validation of server certificates containing internationalized domain names (IDNs) is now supported. As part of this change, the `SSLServer.socket.hostname` attribute now stores the expected hostname in A-label form ("xn--pythn-mua.org"), rather than the U-label form ("pythön.org"). (Contributed by Nathaniel J. Smith and Christian Heimes in bpo-28414.)

The `ssl` module has preliminary and experimental support for TLS 1.3 and OpenSSL 1.1.1. At the time of Python 3.7.0 release, OpenSSL 1.1.1 is still under development and TLS 1.3 hasn't been finalized yet. The TLS 1.3 handshake and protocol behaves slightly differently than TLS 1.2 and earlier, see TLS 1.3. (Contributed by Christian Heimes in bpo-32947, bpo-20995, bpo-29136, bpo-30622 and bpo-33618)

`SSLServer` and `SSLObj` no longer have a public constructor. Direct instantiation was never a documented and supported feature. Instances must be created with `SSLOutput` methods `wrap_socket()` and `wrap_bio()`. (Contributed by Christian Heimes in bpo-32951)

OpenSSL 1.1 APIs for setting the minimum and maximum TLS protocol version are available as `SSLContext.minimum_version` and `SSLContext.maximum_version`. Supported protocols are indicated by several new flags, such as `HAS_TLSv1_1`. (Contributed by Christian Heimes in bpo-32609.)

Added `SSLContext.post_handshake_auth` to enable and `ssl.SSLServer.verify_client_post_handshake()` to initiate TLS 1.3 post-handshake authentication. (Contributed by Christian Heimes in bpo-34670.)
string.Template now lets you to optionally modify the regular expression pattern for braced placeholders and non-braced placeholders separately. (Contributed by Barry Warsaw in bpo-1198569.)

subprocess

The subprocess.run() function accepts the new capture_output keyword argument. When true, stdout and stderr will be captured. This is equivalent to passing subprocess.PIPE as stdout and stderr arguments. (Contributed by Bo Bayles in bpo-32102.)

The subprocess.run function and the subprocess.Popen constructor now accept the text keyword argument as an alias to universal_newlines. (Contributed by Andrew Clegg in bpo-31756.)

On Windows the default for close_fds was changed from False to True when redirecting the standard handles. It's now possible to set close_fds to true when redirecting the standard handles. See subprocess.Popen. This means that close_fds now defaults to True on all supported platforms. (Contributed by Segev Finer in bpo-19764.)

The subprocess module is now more graceful when handling KeyboardInterrupt during subprocess.call(), subprocess.run(), or in a Popen context manager. It now waits a short amount of time for the child to exit, before continuing the handling of the KeyboardInterrupt exception. (Contributed by Gregory P. Smith in bpo-25942.)

sys
The new `sys.breakpointhook()` hook function is called by the built-in `breakpoint()`. (Contributed by Barry Warsaw in bpo-31353.)

On Android, the new `sys.getandroidapilevel()` returns the build-time Android API version. (Contributed by Victor Stinner in bpo-28740.)

The new `sys.get_coroutine_origin_tracking_depth()` function returns the current coroutine origin tracking depth, as set by the new `sys.set_coroutine_origin_tracking_depth()`. `asyncio` has been converted to use this new API instead of the deprecated `sys.set_coroutine_wrapper()`. (Contributed by Nathaniel J. Smith in bpo-32591.)

**time**

PEP 564 adds six new functions with nanosecond resolution to the `time` module:

- `time.clock_gettime_ns()`
- `time.clock_settime_ns()`
- `time.monotonic_ns()`
- `time.perf_counter_ns()`
- `time.process_time_ns()`
- `time.time_ns()`

New clock identifiers have been added:

- `time.CLOCK_BOOTTIME` (Linux): Identical to `time.CLOCK_MONOTONIC`, except it also includes any time that the system is suspended.
- `time.CLOCK_PROF` (FreeBSD, NetBSD and OpenBSD): High-resolution per-process CPU timer.
- `time.CLOCK_UPTIME` (FreeBSD, OpenBSD): Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement.

The new `time.thread_time()` and `time.thread_time_ns()` functions can be used to get per-thread CPU time measurements. (Contributed by Antoine Pitrou in bpo-32025.)

The new `time.pthread_getcpuclockid()` function returns the clock ID of the thread-specific CPU-time clock.

**tkinter**

The new `tkinter.ttk.Spinbox` class is now available. (Contributed by Alan Moore in bpo-32585.)

**tracemalloc**

`tracemalloc.Traceback` behaves more like regular tracebacks, sorting the frames from oldest to most recent. `Traceback.format()` now accepts negative `limit`, truncating the result to the `abs(limit)` oldest frames. To get the old behaviour, use the new `most_recent_first` argument to `Traceback.format()`. (Contributed by Jesse Bakker in bpo-32121.)

**types**

The new `WrapperDescriptorType`, `MethodWrapperType`, `MethodDescriptorType`, and `ClassMethodDescriptorType`
classes are now available. (Contributed by Manuel Krebber and Guido van Rossum in bpo-29377, and Serhiy Storchaka in bpo-32265.)

The new `types.resolve_bases()` function resolves MRO entries dynamically as specified by PEP 560. (Contributed by Ivan Levkivskyi in bpo-32717.)

**unicodedata**

The internal `unicodedata` database has been upgraded to use Unicode 11. (Contributed by Benjamin Peterson.)

**unittest**

The new `-k` command-line option allows filtering tests by a name substring or a Unix shell-like pattern. For example, `python -m unittest -k foo` runs `foo_tests.SomeTest.test_something, bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`. (Contributed by Jonas Haag in bpo-32071.)

**unittest.mock**

The `sentinel` attributes now preserve their identity when they are copied or pickled. (Contributed by Serhiy Storchaka in bpo-20804.)

The new `seal()` function allows sealing `Mock` instances, which will disallow further creation of attribute mocks. The seal is applied recursively to all attributes that are themselves mocks. (Contributed by
urlib.parse

urllib.parse.quote() has been updated from RFC 2396 to RFC 3986, adding ~ to the set of characters that are never quoted by default. (Contributed by Christian Theune and Ratnadeep Debnath in bpo-16285.)

uu

The uu.encode() function now accepts an optional backtick keyword argument. When it's true, zeros are represented by ` instead of spaces. (Contributed by Xiang Zhang in bpo-30103.)

uuid

The new UUID.is_safe attribute relays information from the platform about whether generated UUIDs are generated with a multiprocessing-safe method. (Contributed by Barry Warsaw in bpo-22807.)

uuid.getnode() now prefers universally administered MAC addresses over locally administered MAC addresses. This makes a better guarantee for global uniqueness of UUIDs returned from uuid.uuid1(). If only locally administered MAC addresses are available, the first such one found is returned. (Contributed by Barry Warsaw in bpo-32107.)

warnings

The initialization of the default warnings filters has changed as follows:
warnings enabled via command line options (including those for -b and the new CPython-specific -X dev option) are always passed to the warnings machinery via the sys.warnoptions attribute.

warnings filters enabled via the command line or the environment now have the following order of precedence:

- the BytesWarning filter for -b (or -bb)
- any filters specified with the -W option
- any filters specified with the PYTHONWARNINGS environment variable
- any other CPython specific filters (e.g. the default filter added for the new -X dev mode)
- any implicit filters defined directly by the warnings machinery

in CPython debug builds, all warnings are now displayed by default (the implicit filter list is empty)

(Contributed by Nick Coghlan and Victor Stinner in bpo-20361, bpo-32043, and bpo-32230.)

Deprecation warnings are once again shown by default in single-file scripts and at the interactive prompt. See PEP 565: Show DeprecationWarning in __main__ for details. (Contributed by Nick Coghlan in bpo-31975.)

xml

As mitigation against DTD and external entity retrieval, the xml.dom.minidom and xml.sax modules no longer process external entities by default. (Contributed by Christian Heimes in bpo-
xml.etree

ElementPath predicates in the find() methods can now compare text of the current node with 
[ . = "text"]
, not only text in children. Predicates also allow adding spaces for better readability. (Contributed
by Stefan Behnel in bpo-31648.)

xmlrpc.server

SimpleXMLRPCDispatcher.register_function can now be used as a decorator. (Contributed by Xiang
Zhang in bpo-7769.)

zipapp

Function create_archive() now accepts an optional filter argument to allow the user to select which
files should be included in the archive. (Contributed by Irmen de Jong in bpo-31072.)

Function create_archive() now accepts an optional compressed argument to generate a compressed
archive. A command line option --compress has also been added to support compression. (Contributed
by Zhiming Wang in bpo-31638.)

zipfile

ZipFile now accepts the new compresslevel parameter to control the compression level. (Contributed
by Bo Bayles in bpo-21417.)

Subdirectories in archives created by ZipFile are now stored in
alphabetical order. (Contributed by Bernhard M. Wiedemann in bpo-30693.)
C API Changes

A new API for thread-local storage has been implemented. See PEP 539: New C API for Thread-Local Storage for an overview and Thread Specific Storage (TSS) API for a complete reference. (Contributed by Masayuki Yamamoto in bpo-25658.)

The new context variables functionality exposes a number of new C APIs.

The new PyImport_GetModule() function returns the previously imported module with the given name. (Contributed by Eric Snow in bpo-28411.)

The new Py_RETURN_RICHCOMPARE macro eases writing rich comparison functions. (Contributed by Petr Victorin in bpo-23699.)

The new Py_UNREACHEABLE macro can be used to mark unreachable code paths. (Contributed by Barry Warsaw in bpo-31338.)

The tracemalloc now exposes a C API through the new PyTraceMalloc_Track() and PyTraceMalloc_Untrack() functions. (Contributed by Victor Stinner in bpo-30054.)

The new import__find__load__start() and import__find__load__done() static markers can be used to trace module imports. (Contributed by Christian Heimes in bpo-31574.)

The fields name and doc of structures PyMemberDef, PyGetSetDef, PyStructSequence_Field, PyStructSequence_Desc, and wrapperbase are now of type
const char * rather of char *. (Contributed by Serhiy Storchaka in bpo-28761.)

The result of PyUnicode_AsUTF8AndSize() and PyUnicode_AsUTF8() is now of type const char * rather of char *. (Contributed by Serhiy Storchaka in bpo-28769.)

The result of PyMapping_Keys(), PyMapping_Values() and PyMapping_Items() is now always a list, rather than a list or a tuple. (Contributed by Oren Milman in bpo-28280.)

Added functions PySlice_Unpack() and PySlice_AdjustIndices(). (Contributed by Serhiy Storchaka in bpo-27867.)

PyOS_AfterFork() is deprecated in favour of the new functions PyOS_BeforeFork(), PyOS_AfterFork_Parent() and PyOS_AfterFork_Child(). (Contributed by Antoine Pitrou in bpo-16500.)

The PyExc_RecursionErrorInst singleton that was part of the public API has been removed as its members being never cleared may cause a segfault during finalization of the interpreter. Contributed by Xavier de Gaye in bpo-22898 and bpo-30697.

Added C API support for timezones with timezone constructors PyTimeZone_FromOffset() and PyTimeZone_FromOffsetAndName(), and access to the UTC singleton with PyDateTime_TimeTypeZone_UTC. Contributed by Paul Ganssle in bpo-10381.

The type of results of PyThread_start_new_thread() and
PyThread_get_thread_ident(), and the id parameter of PyThreadState_SetAsyncExc() changed from long to unsigned long. (Contributed by Serhiy Storchaka in bpo-6532.)

PyUnicode_AsWideCharString() now raises a ValueError if the second argument is NULL and the wchar_t* string contains null characters. (Contributed by Serhiy Storchaka in bpo-30708.)

Changes to the startup sequence and the management of dynamic memory allocators mean that the long documented requirement to call Py_Initialize() before calling most C API functions is now relied on more heavily, and failing to abide by it may lead to segfaults in embedding applications. See the Porting to Python 3.7 section in this document and the Before Python Initialization section in the C API documentation for more details.

The new PyInterpreterState_GetID() returns the unique ID for a given interpreter. (Contributed by Eric Snow in bpo-29102.)

Py_DecodeLocale(), Py_EncodeLocale() now use the UTF-8 encoding when the UTF-8 mode is enabled. (Contributed by Victor Stinner in bpo-29240.)

PyUnicode_DecodeLocaleAndSize() and PyUnicode_EncodeLocale() now use the current locale encoding for surrogateescape error handler. (Contributed by Victor Stinner in bpo-29240.)

The start and end parameters of PyUnicode_FindChar() are now adjusted to behave like string slices. (Contributed by Xiang Zhang in bpo-28822.)
Build Changes

Support for building `--without-threads` has been removed. The `threading` module is now always available. (Contributed by Antoine Pitrou in bpo-31370).

A full copy of libffi is no longer bundled for use when building the `_ctypes` module on non-OSX UNIX platforms. An installed copy of libffi is now required when building `_ctypes` on such platforms. (Contributed by Zachary Ware in bpo-27979.)

The Windows build process no longer depends on Subversion to pull in external sources, a Python script is used to download zipfiles from GitHub instead. If Python 3.6 is not found on the system (via `py -3.6`), NuGet is used to download a copy of 32-bit Python for this purpose. (Contributed by Zachary Ware in bpo-30450.)

The `ssl` module requires OpenSSL 1.0.2 or 1.1 compatible libssl. OpenSSL 1.0.1 has reached end of lifetime on 2016-12-31 and is no longer supported. LibreSSL is temporarily not supported as well. LibreSSL releases up to version 2.6.4 are missing required OpenSSL 1.0.2 APIs.
Optimizations

The overhead of calling many methods of various standard library classes implemented in C has been significantly reduced by porting more code to use the METH_FASTCALL convention. (Contributed by Victor Stinner in bpo-29300, bpo-29507, bpo-29452, and bpo-29286.)

Various optimizations have reduced Python startup time by 10% on Linux and up to 30% on macOS. (Contributed by Victor Stinner, INADA Naoki in bpo-29585, and Ivan Levkivskyi in bpo-31333.)

Method calls are now up to 20% faster due to the bytecode changes which avoid creating bound method instances. (Contributed by Yury Selivanov and INADA Naoki in bpo-26110.)

The asyncio module received a number of notable optimizations for commonly used functions:

- The asyncio.get_event_loop() function has been reimplemented in C to make it up to 15 times faster. (Contributed by Yury Selivanov in bpo-32296.)
  
- asyncio.Future callback management has been optimized. (Contributed by Yury Selivanov in bpo-32348.)

- asyncio.gather() is now up to 15% faster. (Contributed by Yury Selivanov in bpo-32355.)

- asyncio.sleep() is now up to 2 times faster when the delay argument is zero or negative. (Contributed by Andrew Svetlov in bpo-32351.)

- The performance overhead of asyncio debug mode has been reduced. (Contributed by Antoine Pitrou in bpo-31970.)

As a result of PEP 560 work, the import time of typing has been reduced by a factor of 7, and many typing operations are now faster.
sorted() and list.sort() have been optimized for common cases to be up to 40-75% faster. (Contributed by Elliot Gorokhovsky in bpo-28685.)

dict.copy() is now up to 5.5 times faster. (Contributed by Yury Selivanov in bpo-31179.)

hasattr() and getattr() are now about 4 times faster when name is not found and obj does not override object.__getattribute__() or object.__getattr__(). (Contributed by INADA Naoki in bpo-32544.)

Searching for certain Unicode characters (like Ukrainian capital “Є”) in a string was up to 25 times slower than searching for other characters. It is now only 3 times slower in the worst case. (Contributed by Serhiy Storchaka in bpo-24821.)

The collections.namedtuple() factory has been reimplemented to make the creation of named tuples 4 to 6 times faster. (Contributed by Jelle Zijlstra with further improvements by INADA Naoki, Serhiy Storchaka, and Raymond Hettinger in bpo-28638.)

date.fromordinal() and date.fromtimestamp() are now up to 30% faster in the common case. (Contributed by Paul Ganssle in bpo-32403.)

The os.fwalk() function is now up to 2 times faster thanks to the use of os.scandir(). (Contributed by Serhiy Storchaka in bpo-25996.)

The speed of the shutil.rmtree() function has been improved by
20–40% thanks to the use of the os.scandir() function. (Contributed by Serhiy Storchaka in bpo-28564.)

Optimized case-insensitive matching and searching of regular expressions. Searching some patterns can now be up to 20 times faster. (Contributed by Serhiy Storchaka in bpo-30285.)

re.compile() now converts flags parameter to int object if it is RegexFlag. It is now as fast as Python 3.5, and faster than Python 3.6 by about 10% depending on the pattern. (Contributed by INADA Naoki in bpo-31671.)

The modify() methods of classes selectors.EpollSelector, selectors.PollSelector and selectors.DevpollSelector may be around 10% faster under heavy loads. (Contributed by Giampaolo Rodola’ in bpo-30014)

Constant folding has been moved from the peephole optimizer to the new AST optimizer, which is able perform optimizations more consistently. (Contributed by Eugene Toder and INADA Naoki in bpo-29469 and bpo-11549.)

Most functions and methods in abc have been rewritten in C. This makes creation of abstract base classes, and calling isinstance() and issubclass() on them 1.5x faster. This also reduces Python start-up time by up to 10%. (Contributed by Ivan Levkivskyi and INADA Naoki in bpo-31333)

Significant speed improvements to alternate constructors for datetime.date and datetime.datetime by using fast-path constructors when not constructing subclasses. (Contributed by Paul Ganssle in bpo-32403)
The speed of comparison of `array.array` instances has been improved considerably in certain cases. It is now from 10x to 70x faster when comparing arrays holding values of the same integer type. (Contributed by Adrian Wielgosik in bpo-24700.)

The `math.erf()` and `math.erfc()` functions now use the (faster) C library implementation on most platforms. (Contributed by Serhiy Storchaka in bpo-26121.)
Other CPython Implementation Changes

- Trace hooks may now opt out of receiving the `line` and opt into receiving the opcode events from the interpreter by setting the corresponding new `f_trace_lines` and `f_trace_opcodes` attributes on the frame being traced. (Contributed by Nick Coghlan in bpo-31344.)
- Fixed some consistency problems with namespace package module attributes. Namespace module objects now have an `__file__` that is set to `None` (previously unset), and their `__spec__.origin` is also set to `None` (previously the string "namespace"). See bpo-32305. Also, the namespace module object's `__spec__.loader` is set to the same value as `__loader__` (previously, the former was set to `None`). See bpo-32303.
- The `locals()` dictionary now displays in the lexical order that variables were defined. Previously, the order was undefined. (Contributed by Raymond Hettinger in bpo-32690.)
- The `distutils upload` command no longer tries to change CR end-of-line characters to CRLF. This fixes a corruption issue with `sdists` that ended with a byte equivalent to CR. (Contributed by Bo Bayles in bpo-32304.)
Deprecated Python Behavior

Yield expressions (both `yield` and `yield from` clauses) are now deprecated in comprehensions and generator expressions (aside from the iterable expression in the leftmost `for` clause). This ensures that comprehensions always immediately return a container of the appropriate type (rather than potentially returning a generator iterator object), while generator expressions won’t attempt to interleave their implicit output with the output from any explicit yield expressions. In Python 3.7, such expressions emit `DeprecationWarning` when compiled, in Python 3.8 this will be a `SyntaxError`. (Contributed by Serhiy Storchaka in bpo-10544.)

Returning a subclass of `complex` from `object.__complex__()` is deprecated and will be an error in future Python versions. This makes `__complex__()`, consistent with `object.__int__()` and `object.__float__()`. (Contributed by Serhiy Storchaka in bpo-28894.)
Deprecated Python modules, functions and methods

aifc

aifc.openfp() has been deprecated and will be removed in Python 3.9. Use aifc.open() instead. (Contributed by Brian Curtin in bpo-31985.)

asyncio

Support for directly await-ing instances of asyncio.Lock and other asyncio synchronization primitives has been deprecated. An asynchronous context manager must be used in order to acquire and release the synchronization resource. (Contributed by Andrew Svetlov in bpo-32253.)

The asyncio.Task.current_task() and asyncio.Task.all_tasks() methods have been deprecated. (Contributed by Andrew Svetlov in bpo-32250.)

collections

In Python 3.8, the abstract base classes in collections.abc will no longer be exposed in the regular collections module. This will help create a clearer distinction between the concrete classes and the abstract base classes. (Contributed by Serhiy Storchaka in bpo-25988.)

dbm
dbm.dumb now supports reading read-only files and no longer writes the index file when it is not changed. A deprecation warning is now emitted if the index file is missing and recreated in the 'r' and 'w' modes (this will be an error in future Python releases). (Contributed by Serhiy Storchaka in bpo-28847.)

**enum**

In Python 3.8, attempting to check for non-Enum objects in Enum classes will raise a `TypeError` (e.g. `1 in Color`); similarly, attempting to check for non-Flag objects in a Flag member will raise `TypeError` (e.g. `1 in Perm.RW`); currently, both operations return `False` instead. (Contributed by Ethan Furman in bpo-33217.)

**gettext**

Using non-integer value for selecting a plural form in `gettext` is now deprecated. It never correctly worked. (Contributed by Serhiy Storchaka in bpo-28692.)

**importlib**

Methods `MetaPathFinder.find_module()` (replaced by `MetaPathFinder.find_spec()`) and `PathEntryFinder.find_loader()` (replaced by `PathEntryFinder.find_spec()`) both deprecated in Python 3.4 now emit `DeprecationWarning`. (Contributed by Matthias Bussonnier in bpo-29576)

The `importlib.abc.ResourceLoader` ABC has been deprecated
in favour of `importlib.abc.ResourceReader`.

locale

`locale.format()` has been deprecated, use `locale.format_string()` instead. (Contributed by Garvit in bpo-10379.)

macpath

The `macpath` is now deprecated and will be removed in Python 3.8. (Contributed by Chi Hsuan Yen in bpo-9850.)

threading

dummy_threading and `_dummy_thread` have been deprecated. It is no longer possible to build Python with threading disabled. Use `threading` instead. (Contributed by Antoine Pitrou in bpo-31370.)

socket

The silent argument value truncation in `socket.htons()` and `socket.ntohs()` has been deprecated. In future versions of Python, if the passed argument is larger than 16 bits, an exception will be raised. (Contributed by Oren Milman in bpo-28332.)

ssl

`ssl.wrap_socket()` is deprecated. Use `ssl.SSLContext.wrap_socket()` instead. (Contributed by
sunau

sunau.openfp() has been deprecated and will be removed in Python 3.9. Use sunau.open() instead. (Contributed by Brian Curtin in bpo-31985.)

sys

Deprecated sys.set_coroutine_wrapper() and sys.get_coroutine_wrapper().

The undocumented sys.callstats() function has been deprecated and will be removed in a future Python version. (Contributed by Victor Stinner in bpo-28799.)

wave

wave.openfp() has been deprecated and will be removed in Python 3.9. Use wave.open() instead. (Contributed by Brian Curtin in bpo-31985.)
Deprecated functions and types of the C API

Function `PySlice_GetIndicesEx()` is deprecated and replaced with a macro if `Py_LIMITED_API` is not set or set to a value in the range between `0x03050400` and `0x03060000` (not inclusive), or is `0x03060100` or higher. (Contributed by Serhiy Storchaka in bpo-27867.)

`PyOS_AfterFork()` has been deprecated. Use `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` or `PyOS_AfterFork_Child()` instead. (Contributed by Antoine Pitrou in bpo-16500.)
Platform Support Removals

- FreeBSD 9 and older are no longer officially supported.

- For full Unicode support, including within extension modules, *nix platforms are now expected to provide at least one of C.UTF-8 (full locale), C.utf8 (full locale) or UTF-8 (LC_CTYPE-only locale) as an alternative to the legacy ASCII-based C locale.

- OpenSSL 0.9.8 and 1.0.1 are no longer supported, which means building CPython 3.7 with SSL/TLS support on older platforms still using these versions requires custom build options that link to a more recent version of OpenSSL.

  Notably, this issue affects the Debian 8 (aka “jessie”) and Ubuntu 14.04 (aka “Trusty”) LTS Linux distributions, as they still use OpenSSL 1.0.1 by default.

  Debian 9 (“stretch”) and Ubuntu 16.04 (“xenial”), as well as recent releases of other LTS Linux releases (e.g. RHEL/CentOS 7.5, SLES 12-SP3), use OpenSSL 1.0.2 or later, and remain supported in the default build configuration.

  CPython's own CI configuration file provides an example of using the SSL compatibility testing infrastructure in CPython's test suite to build and link against OpenSSL 1.1.0 rather than an outdated system provided OpenSSL.
API and Feature Removals

The following features and APIs have been removed from Python 3.7:

- The `os.stat_float_times()` function has been removed. It was introduced in Python 2.3 for backward compatibility with Python 2.2, and was deprecated since Python 3.1.
- Unknown escapes consisting of ' \' and an ASCII letter in replacement templates for `re.sub()` were deprecated in Python 3.5, and will now cause an error.
- Removed support of the `exclude` argument in `tarfile.TarFile.add()`. It was deprecated in Python 2.7 and 3.2. Use the `filter` argument instead.
- The `splitunc()` function in the `ntpath` module was deprecated in Python 3.1, and has now been removed. Use the `splitdrive()` function instead.
- `collections.namedtuple()` no longer supports the `verbose` parameter or `_source` attribute which showed the generated source code for the named tuple class. This was part of an optimization designed to speed-up class creation. (Contributed by Jelle Zijlstra with further improvements by INADA Naoki, Serhiy Storchaka, and Raymond Hettinger in bpo-28638.)
- Functions `bool()`, `float()`, `list()` and `tuple()` no longer take keyword arguments. The first argument of `int()` can now be passed only as positional argument.
- Removed previously deprecated in Python 2.4 classes `Plist`, `Dict` and `_InternalDict` in the `plistlib` module. Dict values in the result of functions `readPlist()` and `readPlistFromBytes()` are now normal dicts. You no longer can use attribute access to access items of these dictionaries.
- The `asyncio.windows_utils.socketpair()` function has been removed. Use the `socket.socketpair()` function
instead, it is available on all platforms since Python 3.5. `asyncio.windows_utils.socketpair` was just an alias to `socket.socketpair` on Python 3.5 and newer.

- **asyncio** no longer exports the `selectors` and `_overlapped` modules as `asyncio.selectors` and `asyncio._overlapped`. Replace from `asyncio` import `selectors` with `import selectors`.

- Direct instantiation of `ssl.SSLSocket` and `ssl.SSLObject` objects is now prohibited. The constructors were never documented, tested, or designed as public constructors. Users were supposed to use `ssl.wrap_socket()` or `ssl.SSLContext`. (Contributed by Christian Heimes in bpo-32951.)

- The unused `distutils install_misc` command has been removed. (Contributed by Eric N. Vander Weele in bpo-29218.)
Module Removals

The `fpectl` module has been removed. It was never enabled by default, never worked correctly on x86-64, and it changed the Python ABI in ways that caused unexpected breakage of C extensions. (Contributed by Nathaniel J. Smith in [bpo-29137](https://bugs.python.org/issue29137).)
Windows-only Changes

The python launcher, (py.exe), can accept 32 & 64 bit specifiers \textbf{without} having to specify a minor version as well. So \texttt{py -3-32} and \texttt{py -3-64} become valid as well as \texttt{py -3.7-32}, also the \texttt{-m-64} and \texttt{-m.n-64} forms are now accepted to force 64 bit python even if 32 bit would have otherwise been used. If the specified version is not available py.exe will error exit. (Contributed by Steve Barnes in \texttt{bpo-30291}.)

The launcher can be run as \texttt{py -0} to produce a list of the installed pythons, \textit{with default marked with an asterisk}. Running \texttt{py -0p} will include the paths. If py is run with a version specifier that cannot be matched it will also print the \textit{short form} list of available specifiers. (Contributed by Steve Barnes in \texttt{bpo-30362}.)
Porting to Python 3.7

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in Python Behavior

- **async** and **await** names are now reserved keywords. Code using these names as identifiers will now raise a *SyntaxError*. (Contributed by Jelle Zijlstra in bpo-30406.)

- **PEP 479** is enabled for all code in Python 3.7, meaning that *StopIteration* exceptions raised directly or indirectly in coroutines and generators are transformed into *RuntimeError* exceptions. (Contributed by Yury Selivanov in bpo-32670.)

- **object.__aiter__()** methods can no longer be declared as asynchronous. (Contributed by Yury Selivanov in bpo-31709.)

- Due to an oversight, earlier Python versions erroneously accepted the following syntax:

```python
f(1 for x in [1],)
class C(1 for x in [1]):
    pass
```

Python 3.7 now correctly raises a *SyntaxError*, as a generator expression always needs to be directly inside a set of parentheses and cannot have a comma on either side, and the duplication of the parentheses can be omitted only on calls. (Contributed by Serhiy Storchaka in bpo-32012 and bpo-32023.)
When using the `-m` switch, the initial working directory is now added to `sys.path`, rather than an empty string (which dynamically denoted the current working directory at the time of each import). Any programs that are checking for the empty string, or otherwise relying on the previous behaviour, will need to be updated accordingly (e.g. by also checking for `os.getcwd()` or `os.path.dirname(__main__.__file__)`, depending on why the code was checking for the empty string in the first place).

Changes in the Python API

- `socketserver.ThreadingMixIn.server_close()` now waits until all non-daemon threads complete. Set the new `socketserver.ThreadingMixIn.block_on_close` class attribute to `False` to get the pre-3.7 behaviour. (Contributed by Victor Stinner in bpo-31233 and bpo-33540.)

- `socketserver.ForkingMixIn.server_close()` now waits until all child processes complete. Set the new `socketserver.ForkingMixIn.block_on_close` class attribute to `False` to get the pre-3.7 behaviour. (Contributed by Victor Stinner in bpo-31151 and bpo-33540.)

- The `locale.localeconv()` function now temporarily sets the `LC_CTYPE` locale to the value of `LC_NUMERIC` in some cases. (Contributed by Victor Stinner in bpo-31900.)

- `pkgutil.walk_packages()` now raises a `ValueError` if `path` is a string. Previously an empty list was returned. (Contributed by Sanyam Khurana in bpo-24744.)

- A format string argument for `string.Formatter.format()` is
now **positional-only**. Passing it as a keyword argument was deprecated in Python 3.5. (Contributed by Serhiy Storchaka in bpo-29193.)

- Attributes **key**, **value** and **coded_value** of class **http.cookies.Morsel** are now read-only. Assigning to them was deprecated in Python 3.5. Use the **set()** method for setting them. (Contributed by Serhiy Storchaka in bpo-29192.)

- The **mode** argument of **os.makedirs()** no longer affects the file permission bits of newly-created intermediate-level directories. To set their file permission bits you can set the umask before invoking **makedirs()**. (Contributed by Serhiy Storchaka in bpo-19930.)

- The **struct.Struct.format** type is now **str** instead of **bytes**. (Contributed by Victor Stinner in bpo-21071.)

- **parse_multipart()** now accepts the **encoding** and **errors** arguments and returns the same results as **FieldStorage**: for non-file fields, the value associated to a key is a list of strings, not bytes. (Contributed by Pierre Quentel in bpo-29979.)

- Due to internal changes in **socket**, calling **socket.fromshare()** on a socket created by **socket.share** in older Python versions is not supported.

- **repr** for **BaseException** has changed to not include the trailing comma. Most exceptions are affected by this change. (Contributed by Serhiy Storchaka in bpo-30399.)

- **repr** for **datetime.timedelta** has changed to include the keyword arguments in the output. (Contributed by Utkarsh
Because `shutil.rmtree()` is now implemented using the `os.scandir()` function, the user specified handler `onerror` is now called with the first argument `os.scandir` instead of `os.listdir` when listing the directory is failed.

Support for nested sets and set operations in regular expressions as in Unicode Technical Standard #18 might be added in the future. This would change the syntax. To facilitate this future change a `FutureWarning` will be raised in ambiguous cases for the time being. That include sets starting with a literal `['` or containing literal character sequences `--`, `&&`, `~~`, and `||`. To avoid a warning, escape them with a backslash. (Contributed by Serhiy Storchaka in bpo-30349.)

The result of splitting a string on a regular expression that could match an empty string has been changed. For example splitting on `r'\s*'` will now split not only on whitespaces as it did previously, but also on empty strings before all non-whitespace characters and just before the end of the string. The previous behavior can be restored by changing the pattern to `r'\s+'`. A `FutureWarning` was emitted for such patterns since Python 3.5.

For patterns that match both empty and non-empty strings, the result of searching for all matches may also be changed in other cases. For example in the string `'a\n\n'`, the pattern `r'(?m)\s*?\$'` will not only match empty strings at positions 2 and 3, but also the string `'\n'` at positions 2–3. To match only blank lines, the pattern should be rewritten as `r'(?m)^[^\S\n]*\$'`.

`re.sub()` now replaces empty matches adjacent to a previous
non-empty match. For example `re.sub('x*', '-', 'abxd')` returns now `-a-b--d-` instead of `-a-b-d-` (the first minus between ‘b’ and ‘d’ replaces ‘x’, and the second minus replaces an empty string between ‘x’ and ‘d’).

(Contributed by Serhiy Storchaka in bpo-25054 and bpo-32308.)

- Change `re.escape()` to only escape regex special characters instead of escaping all characters other than ASCII letters, numbers, and `_`. (Contributed by Serhiy Storchaka in bpo-29995.)

- `tracemalloc.Traceback` frames are now sorted from oldest to most recent to be more consistent with `traceback`. (Contributed by Jesse Bakker in bpo-32121.)

- On OSes that support `socket.SOCK_NONBLOCK` or `socket.SOCK_CLOEXEC` bit flags, the `socket.type` no longer has them applied. Therefore, checks like `if sock.type == socket.SOCK_STREAM` work as expected on all platforms. (Contributed by Yury Selivanov in bpo-32331.)

- On Windows the default for the `close_fds` argument of `subprocess.Popen` was changed from `False` to `True` when redirecting the standard handles. If you previously depended on handles being inherited when using `subprocess.Popen` with standard io redirection, you will have to pass `close_fds=False` to preserve the previous behaviour, or use `STARTUPINFO.lpAttributeList`.

- `importlib.machinery.PathFinder.invalidate_caches()` – which implicitly affects `importlib.invalidate_caches()` –
now deletes entries in `sys.path_importer_cache` which are set to None. (Contributed by Brett Cannon in bpo-33169.)

- In `asyncio`, `loop.sock_recv()`, `loop.sock_sendall()`, `loop.sock_accept()`, `loop.getaddrinfo()` and `loop.getnameinfo()` have been changed to be proper coroutine methods to match their documentation. Previously, these methods returned `asyncio.Future` instances. (Contributed by Yury Selivanov in bpo-32327.)

- `asyncio.Server.sockets` now returns a copy of the internal list of server sockets, instead of returning it directly. (Contributed by Yury Selivanov in bpo-32662.)

- `Struct.format` is now a `str` instance instead of a `bytes` instance. (Contributed by Víctor Stinner in bpo-21071.)

- `ast.literal_eval()` is now stricter. Addition and subtraction of arbitrary numbers are no longer allowed. (Contributed by Serhiy Storchaka in bpo-31778.)

- `Calendar.itermonthdates` will now consistently raise an exception when a date falls outside of the `0001-01-01` through `9999-12-31` range. To support applications that cannot tolerate such exceptions, the new `Calendar.itermonthdays3` and `Calendar.itermonthdays4` can be used. The new methods return tuples and are not restricted by the range supported by `datetime.date`. (Contributed by Alexander Belopolsky in bpo-28292.)

- `collections.ChainMap` now preserves the order of the underlying mappings. (Contributed by Raymond Hettinger in bpo-
The `submit()` method of `concurrent.futures.ThreadPoolExecutor` and `concurrent.futures.ProcessPoolExecutor` now raises a `RuntimeError` if called during interpreter shutdown. (Contributed by Mark Nemec in bpo-33097.)

The `configparser.ConfigParser` constructor now uses `read_dict()` to process the default values, making its behavior consistent with the rest of the parser. Non-string keys and values in the defaults dictionary are now being implicitly converted to strings. (Contributed by James Tocknell in bpo-23835.)

Several undocumented internal imports were removed. One example is that `os.errno` is no longer available; use `import errno` directly instead. Note that such undocumented internal imports may be removed any time without notice, even in micro version releases.

### Changes in the C API

The function `PySlice_GetIndicesEx()` is considered unsafe for resizable sequences. If the slice indices are not instances of `int`, but objects that implement the `__index__() method, the sequence can be resized after passing its length to `PySlice_GetIndicesEx()`. This can lead to returning indices out of the length of the sequence. For avoiding possible problems use new functions `PySlice_Unpack()` and `PySlice_AdjustIndices()`. (Contributed by Serhiy Storchaka in bpo-27867.)
CPython bytecode changes

There are two new opcodes: **LOAD_METHOD** and **CALL_METHOD**. (Contributed by Yury Selivanov and INADA Naoki in [bpo-26110](https://bugs.python.org/issue26110).)

The **STORE_ANNOTATION** opcode has been removed. (Contributed by Mark Shannon in [bpo-32550](https://bugs.python.org/issue32550).)

Windows-only Changes

The file used to override `sys.path` is now called `<python-executable>._pth` instead of `'sys.path'`. See [Finding modules](https://docs.python.org/3/library/sys.html#sys.path) for more information. (Contributed by Steve Dower in [bpo-28137](https://bugs.python.org/issue28137).)

Other CPython implementation changes

In preparation for potential future changes to the public CPython runtime initialization API (see [PEP 432](https://www.python.org/dev/peps/pep-0432/) for an initial, but somewhat outdated, draft), CPython’s internal startup and configuration management logic has been significantly refactored. While these updates are intended to be entirely transparent to both embedding applications and users of the regular CPython CLI, they’re being mentioned here as the refactoring changes the internal order of various operations during interpreter startup, and hence may uncover previously latent defects, either in embedding applications, or in CPython itself. (Initially contributed by Nick Coghlan and Eric Snow as part of [bpo-22257](https://bugs.python.org/issue22257), and further updated by Nick, Eric, and Victor Stinner in a number of other issues). Some known details affected:

- **PySys_AddWarnOptionUnicode()** is not currently usable by embedding applications due to the requirement to create a
Unicode object prior to calling \textit{Py\_Initialize}. Use \texttt{PySys\_AddWarnOption()} instead.

- warnings filters added by an embedding application with \texttt{PySys\_AddWarnOption()} should now more consistently take precedence over the default filters set by the interpreter

Due to changes in the way the default warnings filters are configured, setting \texttt{Py\_BytesWarningFlag} to a value greater than one is no longer sufficient to both emit \texttt{BytesWarning} messages and have them converted to exceptions. Instead, the flag must be set (to cause the warnings to be emitted in the first place), and an explicit \texttt{error::BytesWarning} warnings filter added to convert them to exceptions.

Due to a change in the way docstrings are handled by the compiler, the implicit \texttt{return None} in a function body consisting solely of a docstring is now marked as occurring on the same line as the docstring, not on the function's header line.

The current exception state has been moved from the frame object to the co-routine. This simplified the interpreter and fixed a couple of obscure bugs caused by having swap exception state when entering or exiting a generator. (Contributed by Mark Shannon in \texttt{bpo-25612}.)

Notable changes in Python 3.7.1

Starting in 3.7.1, `Py_Initialize()` now consistently reads and respects all of the same environment settings as `Py_Main()` (in earlier Python versions, it respected an ill-defined subset of those environment variables, while in Python 3.7.0 it didn't read any of them due to bpo-34247). If this behavior is unwanted, set `Py_IgnoreEnvironmentFlag` to 1 before calling `Py_Initialize()`.

In 3.7.1 the C API for Context Variables was updated to use `PyObject` pointers. See also bpo-34762.

`xml.dom.minidom` and `xml.sax` modules no longer process external entities by default. See also bpo-17239.

In 3.7.1 the `tokenize` module now implicitly emits a `NEWLINE` token when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. (Contributed by Ammar Askar in bpo-33899.)
Notable changes in Python 3.7.2

In 3.7.2, `venv` on Windows no longer copies the original binaries, but creates redirector scripts named `python.exe` and `pythonw.exe` instead. This resolves a long standing issue where all virtual environments would have to be upgraded or recreated with each Python update. However, note that this release will still require recreation of virtual environments in order to get the new scripts.
What’s New In Python 3.6

Editors: Elvis Pranskevichus <elvis@magic.io>, Yury Selivanov <yury@magic.io>

This article explains the new features in Python 3.6, compared to 3.5. Python 3.6 was released on December 23, 2016. See the changelog for a full list of changes.

See also: PEP 494 - Python 3.6 Release Schedule
Summary – Release highlights

New syntax features:

- PEP 498, formatted string literals.
- PEP 515, underscores in numeric literals.
- PEP 526, syntax for variable annotations.
- PEP 525, asynchronous generators.
- PEP 530: asynchronous comprehensions.

New library modules:


CPython implementation improvements:

- The dict type has been reimplemented to use a more compact representation based on a proposal by Raymond Hettinger and similar to the PyPy dict implementation. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5.
- Customization of class creation has been simplified with the new protocol.
- The class attribute definition order is now preserved.
- The order of elements in **kwargs now corresponds to the order in which keyword arguments were passed to the function.
- DTrace and SystemTap probing support has been added.
- The new PYTHONMALLOC environment variable can now be used to debug the interpreter memory allocation and access errors.

Significant improvements in the standard library:
The asyncio module has received new features, significant usability and performance improvements, and a fair amount of bug fixes. Starting with Python 3.6 the asyncio module is no longer provisional and its API is considered stable.

A new file system path protocol has been implemented to support path-like objects. All standard library functions operating on paths have been updated to work with the new protocol.

The datetime module has gained support for Local Time Disambiguation.

The typing module received a number of improvements.

The tracemalloc module has been significantly reworked and is now used to provide better output for ResourceWarning as well as provide better diagnostics for memory allocation errors. See the PYTHONMALLOC section for more information.

Security improvements:

- The new secrets module has been added to simplify the generation of cryptographically strong pseudo-random numbers suitable for managing secrets such as account authentication, tokens, and similar.
- On Linux, os.urandom() now blocks until the system urandom entropy pool is initialized to increase the security. See the PEP 524 for the rationale.
- The hashlib and ssl modules now support OpenSSL 1.1.0.
- The default settings and feature set of the ssl module have been improved.
- The hashlib module received support for the BLAKE2, SHA-3 and SHAKE hash algorithms and the scrypt() key derivation function.

Windows improvements:

- PEP 528 and PEP 529, Windows filesystem and console
encoding changed to UTF-8.

- The `py.exe` launcher, when used interactively, no longer prefers Python 2 over Python 3 when the user doesn’t specify a version (via command line arguments or a config file). Handling of shebang lines remains unchanged - “python” refers to Python 2 in that case.

  - `python.exe` and `pythonw.exe` have been marked as long-path aware, which means that the 260 character path limit may no longer apply. See removing the MAX_PATH limitation for details.

  - A `.pth` file can be added to force isolated mode and fully specify all search paths to avoid registry and environment lookup. See the documentation for more information.

  - A `python36.zip` file now works as a landmark to infer `PYTHONHOME`. See the documentation for more information.
New Features

PEP 498: Formatted string literals

PEP 498 introduces a new kind of string literals: f-strings, or formatted string literals.

Formatted string literals are prefixed with 'f' and are similar to the format strings accepted by str.format(). They contain replacement fields surrounded by curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol:

```python
>>> name = "Fred"
>>> f"He said his name is {name}".
'He said his name is Fred.'
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested
'result: 12.35'
```

See also:

PEP 498 – Literal String Interpolation.
PEP written and implemented by Eric V. Smith.

Feature documentation.

PEP 526: Syntax for variable annotations

PEP 484 introduced the standard for type annotations of function
parameters, a.k.a. type hints. This PEP adds syntax to Python for annotating the types of variables including class variables and instance variables:

```python
primes: List[int] = []
captain: str  # Note: no initial value!

class Starship:
    stats: Dict[str, int] = {}
```

Just as for function annotations, the Python interpreter does not attach any particular meaning to variable annotations and only stores them in the `__annotations__` attribute of a class or module.

In contrast to variable declarations in statically typed languages, the goal of annotation syntax is to provide an easy way to specify structured type metadata for third party tools and libraries via the abstract syntax tree and the `__annotations__` attribute.

**See also:**

**PEP 526 – Syntax for variable annotations.**


Tools that use or will use the new syntax: `mypy`, `pytype`, PyCharm, etc.

**PEP 515: Underscores in Numeric Literals**

**PEP 515** adds the ability to use underscores in numeric literals for
improved readability. For example:

```
>>> 1_000_000_000_000_000
1000000000000000
>>> 0x_FF_FF_FF_FF
4294967295
```

Single underscores are allowed between digits and after any base specifier. Leading, trailing, or multiple underscores in a row are not allowed.

The *string formatting* language also now has support for the `_' option to signal the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits:

```
>>> '{:_}'.format(1000000)
'1_000_000'
>>> '{:_x}'.format(0xFFFFFFFF)
'ffff_ffff'
```

See also:

**PEP 515 – Underscores in Numeric Literals**

PEP written by Georg Brandl and Serhiy Storchaka.

**PEP 525: Asynchronous Generators**

**PEP 492** introduced support for native coroutines and `async / await` syntax to Python 3.5. A notable limitation of the Python 3.5 implementation is that it was not possible to use `await` and `yield` in the same function body. In Python 3.6 this restriction has been lifted,
making it possible to define *asynchronous generators*:

```python
async def ticker(delay, to):
    """Yield numbers from 0 to *to* every *delay* seconds.""
    for i in range(to):
        yield i
    await asyncio.sleep(delay)
```

The new syntax allows for faster and more concise code.

**See also:**

**PEP 525** – *Asynchronous Generators*

PEP written and implemented by Yury Selivanov.

**PEP 530**: Asynchronous Comprehensions

**PEP 530** adds support for using `async for` in list, set, dict comprehensions and generator expressions:

```python
result = [i async for i in aiter() if i % 2]
```

Additionally, `await` expressions are supported in all kinds of comprehensions:

```python
result = [await fun() for fun in funcs if await condition]
```

**See also:**

**PEP 530** – *Asynchronous Comprehensions*

PEP written and implemented by Yury Selivanov.
PEP 487: Simpler customization of class creation

It is now possible to customize subclass creation without using a metaclass. The new `__init_subclass__` classmethod will be called on the base class whenever a new subclass is created:

```python
class PluginBase:
    subclasses = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.subclasses.append(cls)

class Plugin1(PluginBase):
    pass

class Plugin2(PluginBase):
    pass
```

In order to allow zero-argument `super()` calls to work correctly from `__init_subclass__()` implementations, custom metaclasses must ensure that the new `__classcell__` namespace entry is propagated to `type.__new__` (as described in Creating the class object).

See also:

**PEP 487 – Simpler customization of class creation**

PEP written and implemented by Martin Teichmann.

Feature documentation

PEP 487: Descriptor Protocol Enhancements
PEP 487 extends the descriptor protocol to include the new optional \_\_set\_name\_\_() method. Whenever a new class is defined, the new method will be called on all descriptors included in the definition, providing them with a reference to the class being defined and the name given to the descriptor within the class namespace. In other words, instances of descriptors can now know the attribute name of the descriptor in the owner class:

```python
class IntField:
    def \_\_get\_\_(self, instance, owner):
        return instance.\_\_dict\_\_[self.name]

    def \_\_set\_\_(self, instance, value):
        if not isinstance(value, int):
            raise ValueError(f'expecting integer in instance.\_\_dict\_\_[self.name] = value

            # this is the new initializer:
    def \_\_set\_name\_\_(self, owner, name):
        self.name = name

class Model:
    int_field = IntField()
```

See also:

**PEP 487 – Simpler customization of class creation**

PEP written and implemented by Martin Teichmann.

**Feature documentation**

PEP 519: Adding a file system path protocol
File system paths have historically been represented as  

`str` or  

`bytes` objects. This has led to people who write code which operate on file system paths to assume that such objects are only one of those two types (an `int` representing a file descriptor does not count as that is not a file path). Unfortunately that assumption prevents alternative object representations of file system paths like `pathlib` from working with pre-existing code, including Python's standard library.

To fix this situation, a new interface represented by `os.PathLike` has been defined. By implementing the `__fspath__()` method, an object signals that it represents a path. An object can then provide a low-level representation of a file system path as a `str` or `bytes` object. This means an object is considered path-like if it implements `os.PathLike` or is a `str` or `bytes` object which represents a file system path. Code can use `os.fspath()`, `os.fsdecode()`, or `os.fsencode()` to explicitly get a `str` and/or `bytes` representation of a path-like object.

The built-in `open()` function has been updated to accept `os.PathLike` objects, as have all relevant functions in the `os` and `os.path` modules, and most other functions and classes in the standard library. The `os.DirEntry` class and relevant classes in `pathlib` have also been updated to implement `os.PathLike`.

The hope is that updating the fundamental functions for operating on file system paths will lead to third-party code to implicitly support all path-like objects without any code changes, or at least very minimal ones (e.g. calling `os.fspath()` at the beginning of code before operating on a path-like object).

Here are some examples of how the new interface allows for `pathlib.Path` to be used more easily and transparently with pre-
existing code:

```python
>>> import pathlib
>>> with open(pathlib.Path("README")) as f:
...     contents = f.read()
...
>>> import os.path
>>> os.path.splitext(pathlib.Path("some_file.txt"))
('some_file', '.txt')
>>> os.path.join("/a/b", pathlib.Path("c"))
'/a/b/c'
>>> import os
>>> os.fspath(pathlib.Path("some_file.txt"))
'some_file.txt'
```

(Implemented by Brett Cannon, Ethan Furman, Dusty Phillips, and Jelle Zijlstra.)

**See also:**

**PEP 519** – Adding a file system path protocol

PEP written by Brett Cannon and Koos Zevenhoven.

**PEP 495**: Local Time Disambiguation

In most world locations, there have been and will be times when local clocks are moved back. In those times, intervals are introduced in which local clocks show the same time twice in the same day. In these situations, the information displayed on a local clock (or stored in a Python `datetime` instance) is insufficient to identify a particular moment in time.

**PEP 495** adds the new `fold` attribute to instances of `datetime.datetime` and `datetime.time` classes to differentiate
between two moments in time for which local times are the same:

```python
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...    u = u0 + i*HOUR
...    t = u.astimezone(Eastern)
...    print(u.time(), 'UTC =', t.time(), t.tzname()
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

The values of the `fold` attribute have the value 0 for all instances except those that represent the second (chronologically) moment in time in an ambiguous case.

**See also:**

**PEP 495 – Local Time Disambiguation**

PEP written by Alexander Belopolsky and Tim Peters, implementation by Alexander Belopolsky.

**PEP 529: Change Windows filesystem encoding to UTF-8**

Representing filesystem paths is best performed with `str` (Unicode) rather than bytes. However, there are some situations where using bytes is sufficient and correct.

Prior to Python 3.6, data loss could result when using bytes paths on Windows. With this change, using bytes to represent paths is now supported on Windows, provided those bytes are encoded with the
encoding returned by `sys.getfilesystemencoding()`, which now defaults to 'utf-8'.

Applications that do not use str to represent paths should use `os.fsencode()` and `os.fsdecode()` to ensure their bytes are correctly encoded. To revert to the previous behaviour, set `PYTHONLEGACYWINDOWSFSENCODING` or call `sys._enablelegacywindowsfsencoding()`.

See [PEP 529](https://www.python.org/dev/peps/pep-0529/) for more information and discussion of code modifications that may be required.

**PEP 528: Change Windows console encoding to UTF-8**

The default console on Windows will now accept all Unicode characters and provide correctly read str objects to Python code. `sys.stdin`, `sys.stdout` and `sys.stderr` now default to utf-8 encoding.

This change only applies when using an interactive console, and not when redirecting files or pipes. To revert to the previous behaviour for interactive console use, set `PYTHONLEGACYWINDOWSSTDIO`.

**See also:**

**PEP 528 – Change Windows console encoding to UTF-8**

PEP written and implemented by Steve Dower.

**PEP 520: Preserving Class Attribute Definition Order**
Attributes in a class definition body have a natural ordering: the same order in which the names appear in the source. This order is now preserved in the new class's `__dict__` attribute.

Also, the effective default class `execution` namespace (returned from `type.__prepare__()`) is now an insertion-order-preserving mapping.

See also:

**PEP 520 – Preserving Class Attribute Definition Order**
PEP written and implemented by Eric Snow.

PEP 468: Preserving Keyword Argument Order

**kwargs in a function signature is now guaranteed to be an insertion-order-preserving mapping.

See also:

**PEP 468 – Preserving Keyword Argument Order**
PEP written and implemented by Eric Snow.

New `dict` implementation

The `dict` type now uses a “compact” representation based on a proposal by Raymond Hettinger which was first implemented by PyPy. The memory usage of the new `dict()` is between 20% and 25% smaller compared to Python 3.5.

The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new `dict`
implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

(Contributed by INADA Naoki in bpo-27350. Idea originally suggested by Raymond Hettinger.)

**PEP 523: Adding a frame evaluation API to CPython**

While Python provides extensive support to customize how code executes, one place it has not done so is in the evaluation of frame objects. If you wanted some way to intercept frame evaluation in Python there really wasn’t any way without directly manipulating function pointers for defined functions.

**PEP 523** changes this by providing an API to make frame evaluation pluggable at the C level. This will allow for tools such as debuggers and JITs to intercept frame evaluation before the execution of Python code begins. This enables the use of alternative evaluation implementations for Python code, tracking frame evaluation, etc.

This API is not part of the limited C API and is marked as private to signal that usage of this API is expected to be limited and only applicable to very select, low-level use-cases. Semantics of the API will change with Python as necessary.

**See also:**

**PEP 523 – Adding a frame evaluation API to CPython**

PEP written by Brett Cannon and Dino Viehland.
The new `PYTHONMALLOC` environment variable allows setting the Python memory allocators and installing debug hooks.

It is now possible to install debug hooks on Python memory allocators on Python compiled in release mode using `PYTHONMALLOC=debug`.

Effects of debug hooks:

- Newly allocated memory is filled with the byte `0xCB`
- Freed memory is filled with the byte `0xDB`
- Detect violations of the Python memory allocator API. For example, `PyObject_Free()` called on a memory block allocated by `PyMem_Malloc()`.
- Detect writes before the start of a buffer (buffer underflows)
- Detect writes after the end of a buffer (buffer overflows)
- Check that the GIL is held when allocator functions of `PYMEM_DOMAIN_OBJC` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.

Checking if the GIL is held is also a new feature of Python 3.6.

See the `PyMem_SetupDebugHooks()` function for debug hooks on Python memory allocators.

It is now also possible to force the usage of the `malloc()` allocator of the C library for all Python memory allocations using `PYTHONMALLOC=malloc`. This is helpful when using external memory debuggers like Valgrind on a Python compiled in release mode.

On error, the debug hooks on Python memory allocators now use the
tracemalloc module to get the traceback where a memory block was allocated.

Example of fatal error on buffer overflow using python3.6 -X tracemalloc=5 (store 5 frames in traces):

Debug memory block at address p=0x7fbc61666f8: API 4 bytes originally requested
The 7 pad bytes at p-7 are FORBIDDENBYTE, as expected.
The 8 pad bytes at tail=0x7fbc61666fc are not all at tail+0: 0x02 *** OUCH
  at tail+1: 0xfb
  at tail+2: 0xfb
  at tail+3: 0xfb
  at tail+4: 0xfb
  at tail+5: 0xfb
  at tail+6: 0xfb
  at tail+7: 0xfb
The block was made by call #1233329 to debug malloc.
Data at p: 1a 2b 30 00

Memory block allocated at (most recent call first):
  File "test/test_bytes.py", line 323
  File "unittest/case.py", line 600
  File "unittest/case.py", line 648
  File "unittest/suite.py", line 122
  File "unittest/suite.py", line 84

Fatal Python error: bad trailing pad byte

Current thread 0x00007fbc6bd32700 (most recent call
  File "test/test_bytes.py", line 323 in test_hex
  File "unittest/case.py", line 600 in run
  File "unittest/case.py", line 648 in __call__
  File "unittest/suite.py", line 122 in run
  File "unittest/suite.py", line 84 in __call__
DTrace and SystemTap probing support

Python can now be built --with-dtrace which enables static markers for the following events in the interpreter:

- function call/return
- garbage collection started/finished
- line of code executed.

This can be used to instrument running interpreters in production, without the need to recompile specific debug builds or providing application-specific profiling/debugging code.

More details in Instrumenting CPython with DTrace and SystemTap.

The current implementation is tested on Linux and macOS. Additional markers may be added in the future.

(Contributed by Łukasz Langa in bpo-21590, based on patches by Jesús Cea Avión, David Malcolm, and Nikhil Benesch.)
Other Language Changes

Some smaller changes made to the core Python language are:

- A **global** or **nonlocal** statement must now textually appear before the first use of the affected name in the same scope. Previously this was a **SyntaxWarning**.
- It is now possible to set a **special method** to **None** to indicate that the corresponding operation is not available. For example, if a class sets `__iter__()` to `None`, the class is not iterable. (Contributed by Andrew Barnert and Ivan Levkivskyi in bpo-25958.)
- Long sequences of repeated traceback lines are now abbreviated as "*[Previous line repeated {count} more times]*" (see traceback for an example). (Contributed by Emanuel Barry in bpo-26823.)
- Import now raises the new exception **ModuleNotFoundError** (subclass of **ImportError**) when it cannot find a module. Code that currently checks for ImportError (in try-except) will still work. (Contributed by Eric Snow in bpo-15767.)
- Class methods relying on zero-argument `super()` will now work correctly when called from metaclass methods during class creation. (Contributed by Martin Teichmann in bpo-23722.)
New Modules

secrets

The main purpose of the new secrets module is to provide an obvious way to reliably generate cryptographically strong pseudo-random values suitable for managing secrets, such as account authentication, tokens, and similar.

**Warning:** Note that the pseudo-random generators in the random module should *NOT* be used for security purposes. Use secrets on Python 3.6+ and os.urandom() on Python 3.5 and earlier.

**See also:**

PEP 506 – Adding A Secrets Module To The Standard Library

PEP written and implemented by Steven D’Aprano.
Improved Modules

array

Exhausted iterators of `array.array` will now stay exhausted even if the iterated array is extended. This is consistent with the behavior of other mutable sequences.

Contributed by Serhiy Storchaka in bpo-26492.

ast

The new `ast.Constant` AST node has been added. It can be used by external AST optimizers for the purposes of constant folding.

Contributed by Victor Stinner in bpo-26146.

asyncio

Starting with Python 3.6 the `asyncio` module is no longer provisional and its API is considered stable.

Notable changes in the `asyncio` module since Python 3.5.0 (all backported to 3.5.x due to the provisional status):

- The `get_event_loop()` function has been changed to always return the currently running loop when called from coroutines and callbacks. (Contributed by Yury Selivanov in bpo-28613.)
- The `ensure_future()` function and all functions that use it, such as `loop.run_until_complete()`, now accept all kinds of `awaitable` objects. (Contributed by Yury Selivanov.)
- New `run_coroutine_threadsafe()` function to submit
coroutines to event loops from other threads. (Contributed by Vincent Michel.)

- New `Transport.is_closing()` method to check if the transport is closing or closed. (Contributed by Yury Selivanov.)
- The `loop.create_server()` method can now accept a list of hosts. (Contributed by Yann Sionneau.)
- New `loop.create_future()` method to create Future objects. This allows alternative event loop implementations, such as uvloop, to provide a faster `asyncio.Future` implementation. (Contributed by Yury Selivanov in bpo-27041.)
- New `loop.get_exception_handler()` method to get the current exception handler. (Contributed by Yury Selivanov in bpo-27040.)
- New `StreamReader.readuntil()` method to read data from the stream until a separator bytes sequence appears. (Contributed by Mark Korenberg.)
- The performance of `StreamReader.readexactly()` has been improved. (Contributed by Mark Korenberg in bpo-28370.)
- The `loop.getaddrinfo()` method is optimized to avoid calling the system `getaddrinfo` function if the address is already resolved. (Contributed by A. Jesse Jiryu Davis.)
- The `loop.stop()` method has been changed to stop the loop immediately after the current iteration. Any new callbacks scheduled as a result of the last iteration will be discarded. (Contributed by Guido van Rossum in bpo-25593.)
- `Future.set_exception` will now raise `TypeError` when passed an instance of the `StopIteration` exception. (Contributed by Chris Angelico in bpo-26221.)
- New `loop.connect_accepted_socket()` method to be used by servers that accept connections outside of asyncio, but that use asyncio to handle them. (Contributed by Jim Fulton in bpo-27392.)
- `TCP_NODELAY` flag is now set for all TCP transports by default.
New `loop.shutdown_asyncgens()` to properly close pending asynchronous generators before closing the loop. (Contributed by Yury Selivanov in bpo-28003.)

**Future** and **Task** classes now have an optimized C implementation which makes asyncio code up to 30% faster. (Contributed by Yury Selivanov and INADA Naoki in bpo-26081 and bpo-28544.)

### binascii

The `b2a_base64()` function now accepts an optional `newline` keyword argument to control whether the newline character is appended to the return value. (Contributed by Victor Stinner in bpo-25357.)

### cmath

The new `cmath.tau` (τ) constant has been added. (Contributed by Lisa Roach in bpo-12345, see [PEP 628](https://www.python.org/dev/peps/pep-0628/) for details.)

New constants: `cmath.inf` and `cmath.nan` to match `math.inf` and `math.nan`, and also `cmath.infj` and `cmath.nanj` to match the format used by complex repr. (Contributed by Mark Dickinson in bpo-23229.)

### collections

The new **Collection** abstract base class has been added to represent sized iterable container classes. (Contributed by Ivan Levkivskyi, docs by Neil Girdhar in bpo-27598.)
The new Reversible abstract base class represents iterable classes that also provide the `__reversed__()` method. (Contributed by Ivan Levkivskyi in bpo-25987.)

The new AsyncGenerator abstract base class represents asynchronous generators. (Contributed by Yury Selivanov in bpo-28720.)

The `namedtuple()` function now accepts an optional keyword argument `module`, which, when specified, is used for the `__module__` attribute of the returned named tuple class. (Contributed by Raymond Hettinger in bpo-17941.)

The `verbose` and `rename` arguments for `namedtuple()` are now keyword-only. (Contributed by Raymond Hettinger in bpo-25628.)

Recursive `collections.deque` instances can now be pickled. (Contributed by Serhiy Storchaka in bpo-26482.)

**concurrent.futures**

The `ThreadPoolExecutor` class constructor now accepts an optional `thread_name_prefix` argument to make it possible to customize the names of the threads created by the pool. (Contributed by Gregory P. Smith in bpo-27664.)

**contextlib**

The `contextlib.AbstractContextManager` class has been added to provide an abstract base class for context managers. It provides a sensible default implementation for `__enter__()` which returns `self` and leaves `__exit__()` an abstract method. A matching
class has been added to the typing module as typing.ContextManager. (Contributed by Brett Cannon in bpo-25609.)

datetime

The datetime and time classes have the new fold attribute used to disambiguate local time when necessary. Many functions in the datetime have been updated to support local time disambiguation. See Local Time Disambiguation section for more information. (Contributed by Alexander Belopolsky in bpo-24773.)

The datetime.strftime() and date.strftime() methods now support ISO 8601 date directives %G, %u and %V. (Contributed by Ashley Anderson in bpo-12006.)

The datetime.isoformat() function now accepts an optional timespec argument that specifies the number of additional components of the time value to include. (Contributed by Alessandro Cucci and Alexander Belopolsky in bpo-19475.)

The datetime.combine() now accepts an optional tzinfo argument. (Contributed by Alexander Belopolsky in bpo-27661.)

decimal

New Decimal.as_integer_ratio() method that returns a pair (n, d) of integers that represent the given Decimal instance as a fraction, in lowest terms and with a positive denominator:

```python
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```
distutils

The `default_format` attribute has been removed from `distutils.command.sdist.sdist` and the `formats` attribute defaults to `['gztar']`. Although not anticipated, any code relying on the presence of `default_format` may need to be adapted. See bpo-27819 for more details.

e-mail

The new e-mail API, enabled via the `policy` keyword to various constructors, is no longer provisional. The e-mail documentation has been reorganized and rewritten to focus on the new API, while retaining the old documentation for the legacy API. (Contributed by R. David Murray in bpo-24277.)

The `email.mime` classes now all accept an optional `policy` keyword. (Contributed by Berker Peksag in bpo-27331.)

The `DecodedGenerator` now supports the `policy` keyword.

There is a new `policy` attribute, `message_factory`, that controls what class is used by default when the parser creates new message objects. For the `email.policy.compat32` policy this is `Message`, for the new policies it is `EmailMessage`. (Contributed by R. David Murray in bpo-20476.)

encodings
On Windows, added the 'oem' encoding to use CP_OEMCP, and the 'ansi' alias for the existing 'mbcs' encoding, which uses the CP_ACP code page. (Contributed by Steve Dower in bpo-27959.)

**enum**

Two new enumeration base classes have been added to the enum module: Flag and IntFlags. Both are used to define constants that can be combined using the bitwise operators. (Contributed by Ethan Furman in bpo-23591.)

Many standard library modules have been updated to use the IntFlags class for their constants.

The new enum.auto value can be used to assign values to enum members automatically:

```python
>>> from enum import Enum, auto
>>> class Color(Enum):
...     red = auto()
...     blue = auto()
...     green = auto()
...
>>> list(Color)
[<Color.red: 1>, <Color.blue: 2>, <Color.green: 3>]
```

**faulthandler**

On Windows, the faulthandler module now installs a handler for Windows exceptions: see faulthandler.enable(). (Contributed by Victor Stinner in bpo-23848.)
fileinput

`hook_encoded()` now supports the `errors` argument. (Contributed by Joseph Hackman in bpo-25788.)

hashlib

`hashlib` supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2. (Contributed by Christian Heimes in bpo-26470.)

BLAKE2 hash functions were added to the module. `blake2b()` and `blake2s()` are always available and support the full feature set of BLAKE2. (Contributed by Christian Heimes in bpo-26798 based on code by Dmitry Chestnykh and Samuel Neves. Documentation written by Dmitry Chestnykh.)

The SHA-3 hash functions `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, and SHAKE hash functions `shake_128()` and `shake_256()` were added. (Contributed by Christian Heimes in bpo-16113. Keccak Code Package by Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer.)

The password-based key derivation function `scrypt()` is now available with OpenSSL 1.1.0 and newer. (Contributed by Christian Heimes in bpo-27928.)

http.client

`HTTPConnection.request()` and `endheaders()` both now support chunked encoding request bodies. (Contributed by Demian
idlelib and IDLE

The idlelib package is being modernized and refactored to make IDLE look and work better and to make the code easier to understand, test, and improve. Part of making IDLE look better, especially on Linux and Mac, is using ttk widgets, mostly in the dialogs. As a result, IDLE no longer runs with tcl/tk 8.4. It now requires tcl/tk 8.5 or 8.6. We recommend running the latest release of either.

‘Modernizing’ includes renaming and consolidation of idlelib modules. The renaming of files with partial uppercase names is similar to the renaming of, for instance, Tkinter and TkFont to tkinter and tkinter.font in 3.0. As a result, imports of idlelib files that worked in 3.5 will usually not work in 3.6. At least a module name change will be needed (see idlelib/README.txt), sometimes more. (Name changes contributed by Al Swiegart and Terry Reedy in bpo-24225. Most idlelib patches since have been and will be part of the process.)

In compensation, the eventual result will be that some idlelib classes will be easier to use, with better APIs and docstrings explaining them. Additional useful information will be added to idlelib when available.

New in 3.6.2:

Multiple fixes for autocompletion. (Contributed by Louie Lu in bpo-15786.)

New in 3.6.3:

Module Browser (on the File menu, formerly called Class Browser), now displays nested functions and classes in addition to top-level
functions and classes. (Contributed by Guilherme Polo, Cheryl Sabella, and Terry Jan Reedy in bpo-1612262.)

The IDLE features formerly implemented as extensions have been reimplemented as normal features. Their settings have been moved from the Extensions tab to other dialog tabs. (Contributed by Charles Wohlganger and Terry Jan Reedy in bpo-27099.)

The Settings dialog (Options, Configure IDLE) has been partly rewritten to improve both appearance and function. (Contributed by Cheryl Sabella and Terry Jan Reedy in multiple issues.)

New in 3.6.4:

The font sample now includes a selection of non-Latin characters so that users can better see the effect of selecting a particular font. (Contributed by Terry Jan Reedy in bpo-13802.) The sample can be edited to include other characters. (Contributed by Serhiy Storchaka in bpo-31860.)

New in 3.6.6:

Editor code context option revised. Box displays all context lines up to maxlines. Clicking on a context line jumps the editor to that line. Context colors for custom themes is added to Highlights tab of Settings dialog. (Contributed by Cheryl Sabella and Terry Jan Reedy in bpo-33642, bpo-33768, and bpo-33679.)

On Windows, a new API call tells Windows that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect. (Contributed by Terry Jan Reedy in bpo-33656.)
New in 3.6.7:

Output over N lines (50 by default) is squeezed down to a button. N can be changed in the PyShell section of the General page of the Settings dialog. Fewer, but possibly extra long, lines can be squeezed by right clicking on the output. Squeezed output can be expanded in place by double-clicking the button or into the clipboard or a separate window by right-clicking the button. (Contributed by Tal Einat in bpo-1529353.)

importlib

Import now raises the new exception ModuleNotFoundError (subclass of ImportError) when it cannot find a module. Code that current checks for ImportError (in try-except) will still work. (Contributed by Eric Snow in bpo-15767.)

importlib.util.LazyLoader now calls create_module() on the wrapped loader, removing the restriction that importlib.machinery.BuiltinImporter and importlib.machinery.ExtensionFileLoader couldn't be used with importlib.util.LazyLoader.

importlib.util.cache_from_source(), importlib.util.source_from_cache(), and importlib.util.spec_from_file_location() now accept a path-like object.

inspect

The inspect.signature() function now reports the implicit
parameters generated by the compiler for comprehension and generator expression scopes as if they were positional-only parameters called `implicit0`. (Contributed by Jelle Zijlstra in bpo-19611.)

To reduce code churn when upgrading from Python 2.7 and the legacy `inspect.getargspec()` API, the previously documented deprecation of `inspect.getfullargspec()` has been reversed. While this function is convenient for single/source Python 2/3 code bases, the richer `inspect.signature()` interface remains the recommended approach for new code. (Contributed by Nick Coghlan in bpo-27172)

json

`json.load()` and `json.loads()` now support binary input. Encoded JSON should be represented using either UTF-8, UTF-16, or UTF-32. (Contributed by Serhiy Storchaka in bpo-17909.)

logging

The new `WatchedFileHandler.reopenIfNeeded()` method has been added to add the ability to check if the log file needs to be reopened. (Contributed by Marian Horban in bpo-24884.)

math

The tau (τ) constant has been added to the `math` and `cmath` modules. (Contributed by Lisa Roach in bpo-12345, see PEP 628 for details.)
multiprocessing

Proxy Objects returned by multiprocessing.Manager() can now be nested. (Contributed by Davin Potts in bpo-6766.)

OS

See the summary of PEP 519 for details on how the os and os.path modules now support path-like objects.

scandir() now supports bytes paths on Windows.

A new close() method allows explicitly closing a scandir() iterator. The scandir() iterator now supports the context manager protocol. If a scandir() iterator is neither exhausted nor explicitly closed a ResourceWarning will be emitted in its destructor. (Contributed by Serhiy Storchaka in bpo-25994.)

On Linux, os.urandom() now blocks until the system urandom entropy pool is initialized to increase the security. See the PEP 524 for the rationale.

The Linux getrandom() syscall (get random bytes) is now exposed as the new os.getrandom() function. (Contributed by Victor Stinner, part of the PEP 524)

pathlib

pathlib now supports path-like objects. (Contributed by Brett Cannon in bpo-27186.)
See the summary of PEP 519 for details.

**pdb**

The `Pdb` class constructor has a new optional `readrc` argument to control whether `.pdbrc` files should be read.

**pickle**

Objects that need `__new__` called with keyword arguments can now be pickled using `pickle protocols` older than protocol version 4. Protocol version 4 already supports this case. (Contributed by Serhiy Storchaka in bpo-24164.)

**pickletools**

`pickletools.dis()` now outputs the implicit memo index for the `MEMOIZE` opcode. (Contributed by Serhiy Storchaka in bpo-25382.)

**pydoc**

The `pydoc` module has learned to respect the `MANPAGER` environment variable. (Contributed by Matthias Klose in bpo-8637.)

`help()` and `pydoc` can now list named tuple fields in the order they were defined rather than alphabetically. (Contributed by Raymond Hettinger in bpo-24879.)

**random**
The new `choices()` function returns a list of elements of specified size from the given population with optional weights. (Contributed by Raymond Hettinger in bpo-18844.)

**re**

Added support of modifier spans in regular expressions. Examples: `'(?!p)ython'` matches `python` and `Python`, but not `PYTHON`; `'(?!g(?-i:v)r)\r` matches `GvR` and `gvr`, but not `GVR`. (Contributed by Serhiy Storchaka in bpo-433028.)

Match object groups can be accessed by `__getitem__`, which is equivalent to `group()`. So `mo['name']` is now equivalent to `mo.group('name')`. (Contributed by Eric Smith in bpo-24454.)

Match objects now support **index-like objects** as group indices. (Contributed by Jeroen Demeyer and Xiang Zhang in bpo-27177.)

**readline**

Added `set_auto_history()` to enable or disable automatic addition of input to the history list. (Contributed by Tyler Crompton in bpo-26870.)

**rlcompleter**

Private and special attribute names now are omitted unless the prefix starts with underscores. A space or a colon is added after some completed keywords. (Contributed by Serhiy Storchaka in bpo-25011 and bpo-25209.)
shlex

The shlex has much improved shell compatibility through the new punctuation_chars argument to control which characters are treated as punctuation. (Contributed by Vinay Sajip in bpo-1521950.)
site

When specifying paths to add to sys.path in a .pth file, you may now specify file paths on top of directories (e.g. zip files). (Contributed by Wolfgang Langner in bpo-26587).

sqlite3

sqlite3.Cursor.lastrowid now supports the REPLACE statement. (Contributed by Alex LordThorsen in bpo-16864.)

socket

The ioctl() function now supports the SIO_LOOPBACK_FAST_PATH control code. (Contributed by Daniel Stokes in bpo-26536.)

The getssockopt() constants SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, and SO_PASSSEC are now supported. (Contributed by Christian Heimes in bpo-26907.)

The setsockopt() now supports the setsockopt(level, optname, None, optlen: int) form. (Contributed by Christian Heimes in bpo-27744.)
The socket module now supports the address family **AF_ALG** to interface with Linux Kernel crypto API. **ALG_***, **SOL_ALG** and **sendmsg_afalg()** were added. (Contributed by Christian Heimes in bpo-27744 with support from Victor Stinner.)

New Linux constants **TCP_USER_TIMEOUT** and **TCP_CONGESTION** were added. (Contributed by Omar Sandoval, issue:26273).

**socketserver**

Servers based on the **socketserver** module, including those defined in **http.server**, **xmlrpc.server** and **wsgiref.simple_server**, now support the context manager protocol. (Contributed by Aviv Palivoda in bpo-26404.)

The **wfile** attribute of **StreamRequestHandler** classes now implements the **io.BufferedIOBase** writable interface. In particular, calling **write()** is now guaranteed to send the data in full. (Contributed by Martin Panter in bpo-26721.)

**ssl**

**ssl** supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2. (Contributed by Christian Heimes in bpo-26470.)

3DES has been removed from the default cipher suites and ChaCha20 Poly1305 cipher suites have been added. (Contributed by Christian Heimes in bpo-27850 and bpo-27766.)

**SSLContext** has better default configuration for options and ciphers. (Contributed by Christian Heimes in bpo-28043.)
SSL session can be copied from one client-side connection to another with the new SSLSession class. TLS session resumption can speed up the initial handshake, reduce latency and improve performance (Contributed by Christian Heimes in bpo-19500 based on a draft by Alex Warhawk.)

The new get_ciphers() method can be used to get a list of enabled ciphers in order of cipher priority.

All constants and flags have been converted to IntEnum and IntFlags. (Contributed by Christian Heimes in bpo-28025.)

Server and client-side specific TLS protocols for SSLContext were added. (Contributed by Christian Heimes in bpo-28085.)

statistics

A new harmonic_mean() function has been added. (Contributed by Steven D'Aprano in bpo-27181.)

struct

struct now supports IEEE 754 half-precision floats via the 'e' format specifier. (Contributed by Eli Stevens, Mark Dickinson in bpo-11734.)

subprocess

subprocess.Popen destructor now emits a ResourceWarning warning if the child process is still running. Use the context manager protocol (with proc: ...) or explicitly call the wait() method to
read the exit status of the child process. (Contributed by Victor Stinner in bpo-26741.)

The subprocess.Popen constructor and all functions that pass arguments through to it now accept encoding and errors arguments. Specifying either of these will enable text mode for the stdin, stdout and stderr streams. (Contributed by Steve Dower in bpo-6135.)

**sys**

The new getfilesystemencodeerrors() function returns the name of the error mode used to convert between Unicode filenames and bytes filenames. (Contributed by Steve Dower in bpo-27781.)

On Windows the return value of the getwindowsversion() function now includes the platform_version field which contains the accurate major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process (Contributed by Steve Dower in bpo-27932.)

**telnetlib**

Telnet is now a context manager (contributed by Stéphane Wirtel in bpo-25485).

**time**

The struct_time attributes tm_gmtoff and tm_zone are now available on all platforms.
The new `Timer.autorange()` convenience method has been added to call `Timer.timeit()` repeatedly so that the total run time is greater or equal to 200 milliseconds. (Contributed by Steven D’Aprano in `bpo-6422`.)

`timeit` now warns when there is substantial (4x) variance between best and worst times. (Contributed by Serhiy Storchaka in `bpo-23552`.)

tkinter

Added methods `trace_add()`, `trace_remove()` and `trace_info()` in the `tkinter.Variable` class. They replace old methods `trace_variable()`, `trace()`, `trace_vdelete()` and `trace_vinfo()` that use obsolete Tcl commands and might not work in future versions of Tcl. (Contributed by Serhiy Storchaka in `bpo-22115`.)

traceback

Both the traceback module and the interpreter’s builtin exception display now abbreviate long sequences of repeated lines in tracebacks as shown in the following example:

```python
>>> def f(): f()
... f()
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in f
  File "<stdin>", line 1, in f
  ... [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```
tracemalloc

The tracemalloc module now supports tracing memory allocations in multiple different address spaces.

The new DomainFilter filter class has been added to filter block traces by their address space (domain).

typing

Since the typing module is provisional, all changes introduced in Python 3.6 have also been backported to Python 3.5.x.

The typing module has a much improved support for generic type aliases. For example Dict[str, Tuple[S, T]] is now a valid type annotation. (Contributed by Guido van Rossum in Github #195.)

The typing.ContextManager class has been added for representing contextlib.AbstractContextManager. (Contributed by Brett Cannon in bpo-25609.)

The typing.Collection class has been added for representing collections.abc.Collection. (Contributed by Ivan Levkivskyi in bpo-27598.)

The typing.ClassVar type construct has been added to mark class variables. As introduced in PEP 526, a variable annotation wrapped in
ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. (Contributed by Ivan Levkivskyi in Github #280.)

A new `TYPE_CHECKING` constant that is assumed to be `True` by the static type checkers, but is `False` at runtime. (Contributed by Guido van Rossum in Github #230.)

A new `NewType()` helper function has been added to create lightweight distinct types for annotations:

```python
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. (Contributed by Ivan Levkivskyi in Github #189.)

`unicodedata`

The `unicodedata` module now uses data from Unicode 9.0.0. (Contributed by Benjamin Peterson.)

`unittest.mock`

The `Mock` class has the following improvements:

- Two new methods, `Mock.assert_called()` and `Mock.assert_called_once()` to check if the mock object was called. (Contributed by Amit Saha in bpo-26323.)
- The `Mock.reset_mock()` method now has two optional keyword only arguments: `return_value` and `side_effect`. 
urllib.request

If a HTTP request has a file or iterable body (other than a bytes object) but no `Content-Length` header, rather than throwing an error, `AbstractHTTPHandler` now falls back to use chunked transfer encoding. (Contributed by Demian Brecht and Rolf Krah in `bpo-12319`.)

urllib.robotparser

`RobotFileParser` now supports the `Crawl-delay` and `Request-rate` extensions. (Contributed by Nikolay Bogoychev in `bpo-16099`.)

venv

`venv` accepts a new parameter `-p`. This parameter provides an alternative prefix for the virtual environment. (Proposed by Łukasz Balcerzak and ported to 3.6 by Stéphane Wirtel in `bpo-22829`.)

warnings

A new optional `source` parameter has been added to the `warnings.warn_explicit()` function: the destroyed object which emitted a `ResourceWarning`. A `source` attribute has also been added to `warnings.WarningMessage` (contributed by Victor Stinner in `bpo-26568` and `bpo-26567`).

When a `ResourceWarning` warning is logged, the `tracemalloc` module is now used to try to retrieve the traceback where the
destroyed object was allocated.

Example with the script example.py:

```python
import warnings

def func():
    return open(__file__)

f = func()
f = None
```

Output of the command `python3.6 -Wd -X tracemalloc=5 example.py`:

```
example.py:7: ResourceWarning: unclosed file <_io.TextIOWrapper '<__main__.example'> mode: 'r', closefd: True> at most recently closed file
  f = None

Object allocated at (most recent call first):
  File "example.py", lineno 4
    return open(__file__)
  File "example.py", lineno 6
    f = func()
```

The “Object allocated at” traceback is new and is only displayed if `tracemalloc` is tracing Python memory allocations and if the `warnings` module was already imported.

**winreg**

Added the 64-bit integer type `REG_QWORD`. (Contributed by Clement Rouault in bpo-23026.)

**winsound**
Allowed keyword arguments to be passed to \texttt{Beep}, \texttt{MessageBeep}, and \texttt{PlaySound} (bpo-27982).

\texttt{xmlrpc.client}

The \texttt{xmlrpc.client} module now supports unmarshalling additional data types used by the Apache XML-RPC implementation for numerics and \texttt{None}. (Contributed by Serhiy Storchaka in bpo-26885.)

\texttt{zipfile}

A new \texttt{ZipInfo.from\_file()} class method allows making a \texttt{ZipInfo} instance from a filesystem file. A new \texttt{ZipInfo.is\_dir()} method can be used to check if the \texttt{ZipInfo} instance represents a directory. (Contributed by Thomas Kluyver in bpo-26039.)

The \texttt{ZipFile.open()} method can now be used to write data into a ZIP file, as well as for extracting data. (Contributed by Thomas Kluyver in bpo-26039.)

\texttt{zlib}

The \texttt{compress()} and \texttt{decompress()} functions now accept keyword arguments. (Contributed by Aviv Palivoda in bpo-26243 and Xiang Zhang in bpo-16764 respectively.)
Optimizations

- The Python interpreter now uses a 16-bit wordcode instead of bytecode which made a number of opcode optimizations possible. (Contributed by Demur Rumed with input and reviews from Serhiy Storchaka and Victor Stinner in bpo-26647 and bpo-28050.)
- The `asyncio.Future` class now has an optimized C implementation. (Contributed by Yury Selivanov and INADA Naoki in bpo-26081.)
- The `asyncio.Task` class now has an optimized C implementation. (Contributed by Yury Selivanov in bpo-28544.)
- Various implementation improvements in the `typing` module (such as caching of generic types) allow up to 30 times performance improvements and reduced memory footprint.
- The ASCII decoder is now up to 60 times as fast for error handlers `surrogateescape`, `ignore` and `replace` (Contributed by Victor Stinner in bpo-24870).
- The ASCII and the Latin1 encoders are now up to 3 times as fast for the error handler `surrogateescape` (Contributed by Victor Stinner in bpo-25227).
- The UTF-8 encoder is now up to 75 times as fast for error handlers `ignore`, `replace`, `surrogateescape`, `surrogatepass` (Contributed by Victor Stinner in bpo-25267).
- The UTF-8 decoder is now up to 15 times as fast for error handlers `ignore`, `replace` and `surrogateescape` (Contributed by Victor Stinner in bpo-25301).
- `bytes % args` is now up to 2 times faster. (Contributed by Victor Stinner in bpo-25349).
- `bytearray % args` is now between 2.5 and 5 times faster. (Contributed by Victor Stinner in bpo-25399).
- Optimize `bytes.fromhex()` and `bytearray.fromhex()`: they are now between 2x and 3.5x faster. (Contributed by Victor
Stinner in bpo-25401).

- Optimize `bytes.replace(b'', b'.')` and `bytearray.replace(b'', b'.')`: up to 80% faster. (Contributed by Josh Snider in bpo-26574).

- Allocator functions of the PyMem_Malloc() domain (PYMEM_DOMAIN_MEM) now use the pymalloc memory allocator instead of malloc() function of the C library. The pymalloc allocator is optimized for objects smaller or equal to 512 bytes with a short lifetime, and use malloc() for larger memory blocks. (Contributed by Victor Stinner in bpo-26249).

- `pickle.load()` and `pickle.loads()` are now up to 10% faster when deserializing many small objects (Contributed by Victor Stinner in bpo-27056).

- Passing keyword arguments to a function has an overhead in comparison with passing positional arguments. Now in extension functions implemented with using Argument Clinic this overhead is significantly decreased. (Contributed by Serhiy Storchaka in bpo-27574).

- Optimized `glob()` and `iglob()` functions in the `glob` module; they are now about 3–6 times faster. (Contributed by Serhiy Storchaka in bpo-25596).

- Optimized globbing in `pathlib` by using `os.scandir()`; it is now about 1.5–4 times faster. (Contributed by Serhiy Storchaka in bpo-26032).

- `xml.etree.ElementTree` parsing, iteration and deepcopy performance has been significantly improved. (Contributed by Serhiy Storchaka in bpo-25638, bpo-25873, and bpo-25869.)

- Creation of `fractions.Fraction` instances from floats and decimals is now 2 to 3 times faster. (Contributed by Serhiy Storchaka in bpo-25971.)
Build and C API Changes

- Python now requires some C99 support in the toolchain to build. Most notably, Python now uses standard integer types and macros in place of custom macros like `PY_LONG_LONG`. For more information, see PEP 7 and bpo-17884.
- Cross-compiling CPython with the Android NDK and the Android API level set to 21 (Android 5.0 Lollipop) or greater runs successfully. While Android is not yet a supported platform, the Python test suite runs on the Android emulator with only about 16 tests failures. See the Android meta-issue bpo-26865.
- The `--enable-optimizations` configure flag has been added. Turning it on will activate expensive optimizations like PGO. (Original patch by Alecsandru Patrascu of Intel in bpo-26359.)
- The GIL must now be held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.
- New `Py_FinalizeEx()` API which indicates if flushing buffered data failed. (Contributed by Martin Panter in bpo-5319.)
- `PyArg_ParseTupleAndKeywords()` now supports positional-only parameters. Positional-only parameters are defined by empty names. (Contributed by Serhiy Storchaka in bpo-26282).
- `PyTraceback_Print` method now abbreviates long sequences of repeated lines as "[Previous line repeated {count} more times]". (Contributed by Emanuel Barry in bpo-26823.)
- The new `PyErr_SetImportErrorSubclass()` function allows for specifying a subclass of `ImportError` to raise. (Contributed by Eric Snow in bpo-15767.)
- The new `PyErr_ResourceWarning()` function can be used to generate a `ResourceWarning` providing the source of the resource allocation. (Contributed by Victor Stinner in bpo-26567.)
The new `PyOS_FSPath()` function returns the file system representation of a path-like object. (Contributed by Brett Cannon in bpo-27186.)

The `PyUnicode_FSConverter()` and `PyUnicode_FSDecoder()` functions will now accept path-like objects.
Other Improvements

- When **--version** (short form: **-V**) is supplied twice, Python prints **sys.version** for detailed information.

```
$ ./python -VV
Python 3.6.0b4+ (3.6:223967b49e49+, Nov 21 2016, [GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)]
```
Deprecated

New Keywords

async and await are not recommended to be used as variable, class, function or module names. Introduced by PEP 492 in Python 3.5, they will become proper keywords in Python 3.7. Starting in Python 3.6, the use of async or await as names will generate a DeprecationWarning.

Deprecated Python behavior

Raising the StopIteration exception inside a generator will now generate a DeprecationWarning, and will trigger a RuntimeError in Python 3.7. See PEP 479: Change StopIteration handling inside generators for details.

The __aiter__() method is now expected to return an asynchronous iterator directly instead of returning an awaitable as previously. Doing the former will trigger a DeprecationWarning. Backward compatibility will be removed in Python 3.7. (Contributed by Yury Selivanov in bpo-27243.)

A backslash-character pair that is not a valid escape sequence now generates a DeprecationWarning. Although this will eventually become a SyntaxError, that will not be for several Python releases. (Contributed by Emanuel Barry in bpo-27364.)

When performing a relative import, falling back on __name__ and __path__ from the calling module when __spec__ or __package__ are not defined now raises an ImportWarning.
(Contributed by Rose Ames in bpo-25791.)

**Deprecated Python modules, functions and methods**

**asynchat**

The `asynchat` has been deprecated in favor of `asyncio`. (Contributed by Mariatta in bpo-25002.)

**asyncore**

The `asyncore` has been deprecated in favor of `asyncio`. (Contributed by Mariatta in bpo-25002.)

**dbm**

Unlike other `dbm` implementations, the `dbm.dumb` module creates databases with the `rw` mode and allows modifying the database opened with the `'r'` mode. This behavior is now deprecated and will be removed in 3.8. (Contributed by Serhiy Storchaka in bpo-21708.)

**distutils**

The undocumented `extra_path` argument to the Distribution constructor is now considered deprecated and will raise a warning if set. Support for this parameter will be removed in a future Python release. See bpo-27919 for details.

**grp**

The support of non-integer arguments in `getgrgid()` has been
deprecated. (Contributed by Serhiy Storchaka in bpo-26129.)

importlib

The `importlib.machinery.SourceFileLoader.load_module()` and `importlib.machinery.SourcelessFileLoader.load_module` methods are now deprecated. They were the only remaining implementations of `importlib.abc.Loader.load_module()` in `importlib` that had not been deprecated in previous versions of Python in favour of `importlib.abc.Loader.exec_module()`.

The `importlib.machinery.WindowsRegistryFinder` class is now deprecated. As of 3.6.0, it is still added to `sys.meta_path` by default (on Windows), but this may change in future releases.

os

Undocumented support of general `bytes-like objects` as paths in `os` functions, `compile()` and similar functions is now deprecated. (Contributed by Serhiy Storchaka in bpo-25791 and bpo-26754.)

re

Support for inline flags `(?letters)` in the middle of the regular expression has been deprecated and will be removed in a future Python version. Flags at the start of a regular expression are still allowed. (Contributed by Serhiy Storchaka in bpo-22493.)

ssl
OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.

SSL-related arguments like certfile, keyfile and check_hostname in ftplib, http.client, imaplib, poplib, and smtplib have been deprecated in favor of context. (Contributed by Christian Heimes in bpo-28022.)

A couple of protocols and functions of the ssl module are now deprecated. Some features will no longer be available in future versions of OpenSSL. Other features are deprecated in favor of a different API. (Contributed by Christian Heimes in bpo-28022 and bpo-26470.)

tkinter

The tkinter.tix module is now deprecated. tkinter users should use tkinter.ttk instead.

venv

The pyvenv script has been deprecated in favour of python3 -m venv. This prevents confusion as to what Python interpreter pyvenv is connected to and thus what Python interpreter will be used by the virtual environment. (Contributed by Brett Cannon in bpo-25154.)

Deprecated functions and types of the C API

Undocumented functions PyUnicode_AsEncodedObject(), PyUnicode_AsDecodedObject(),
PyUnicode_AsEncodedUnicode() and PyUnicode_AsDecodedUnicode() are deprecated now. Use the generic codec based API instead.

Deprecated Build Options

The `--with-system-ffi` configure flag is now on by default on non-macOS UNIX platforms. It may be disabled by using `--without-system-ffi`, but using the flag is deprecated and will not be accepted in Python 3.7. macOS is unaffected by this change. Note that many OS distributors already use the `--with-system-ffi` flag when building their system Python.
Removed

API and Feature Removals

- Unknown escapes consisting of '\' and an ASCII letter in regular expressions will now cause an error. In replacement templates for `re.sub()` they are still allowed, but deprecated. The `re.LOCALE` flag can now only be used with binary patterns.
- `inspect.getmoduleinfo()` was removed (was deprecated since CPython 3.3). `inspect.getmodulename()` should be used for obtaining the module name for a given path. (Contributed by Yury Selivanov in bpo-13248.)
- `traceback.Ignore` class and `traceback.usage`, `traceback.modname`, `traceback.fullmodname`, `traceback.find_lines_from_code`, `traceback.find_lines`, `traceback.find_strings`, `traceback.find_executable_lines` methods were removed from the `traceback` module. They were undocumented methods deprecated since Python 3.2 and equivalent functionality is available from private methods.
- The `tk_menuBar()` and `tk_bindForTraversal()` dummy methods in `tkinter` widget classes were removed (corresponding Tk commands were obsolete since Tk 4.0).
- The `open()` method of the `zipfile.ZipFile` class no longer supports the 'U' mode (was deprecated since Python 3.4). Use `io.TextIOWrapper` for reading compressed text files in `universal newlines` mode.
- The undocumented `IN`, `CDROM`, `DLFCN`, `TYPES`, `CDIO`, and `STROPTS` modules have been removed. They had been available in the platform specific `Lib/plat-*` directories, but were chronically out of date, inconsistently available across platforms, and unmaintained. The script that created these modules is still
available in the source distribution at Tools/scripts/h2py.py.
- The deprecated `asynchat.fifo` class has been removed.
Porting to Python 3.6

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in ‘python’ Command Behavior

- The output of a special Python build with defined `COUNT_ALLOCS`, `SHOW_ALLOC_COUNT` or `SHOW_TRACK_COUNT` macros is now off by default. It can be re-enabled using the `-X showallococcount` option. It now outputs to `stderr` instead of `stdout`. (Contributed by Serhiy Storchaka in [bpo-23034](https://bugs.python.org/issue23034).)

Changes in the Python API

- `open()` will no longer allow combining the `'U'` mode flag with `'+'`. (Contributed by Jeff Balogh and John O'Connor in [bpo-2091](https://bugs.python.org/issue2091).)

- `sqlite3` no longer implicitly commits an open transaction before DDL statements.

- On Linux, `os.urandom()` now blocks until the system urandom entropy pool is initialized to increase the security.

- When `importlib.abc.Loader.exec_module()` is defined, `importlib.abc.Loader.create_module()` must also be defined.

- `PyErr_SetImportError()` now sets `TypeError` when its `msg` argument is not set. Previously only `NULL` was returned.
• The format of the `co_lnotab` attribute of code objects changed to support a negative line number delta. By default, Python does not emit bytecode with a negative line number delta. Functions using `frame.f_lineno`, `PyFrame_GetLineNumber()` or `PyCode_Addr2Line()` are not affected. Functions directly decoding `co_lnotab` should be updated to use a signed 8-bit integer type for the line number delta, but this is only required to support applications using a negative line number delta. See `Objects/lnotab_notes.txt` for the `co_lnotab` format and how to decode it, and see the PEP 511 for the rationale.

• The functions in the `compileall` module now return booleans instead of 1 or 0 to represent success or failure, respectively. Thanks to booleans being a subclass of integers, this should only be an issue if you were doing identity checks for 1 or 0. See bpo-25768.

• Reading the port attribute of `urllib.parse.urlsplit()` and `urlparse()` results now raises `ValueError` for out-of-range values, rather than returning `None`. See bpo-20059.

• The `imp` module now raises a `DeprecationWarning` instead of `PendingDeprecationWarning`.

• The following modules have had missing APIs added to their `__all__` attributes to match the documented APIs: `calendar`, `cgi`, `csv`, `ElementTree`, `enum`, `fileinput`, `ftplib`, `logging`, `mailbox`, `mimetypes`, `optparse`, `plistlib`, `smtpd`, `subprocess`, `tarfile`, `threading` and `wave`. This means they will export new symbols when `import *` is used. (Contributed by Joel Taddei and Jacek Kołodziej in bpo-23883.)
• When performing a relative import, if `__package__` does not compare equal to `__spec__.parent` then `ImportWarning` is raised. (Contributed by Brett Cannon in bpo-25791.)

• When a relative import is performed and no parent package is known, then `ImportError` will be raised. Previously, `SystemError` could be raised. (Contributed by Brett Cannon in bpo-18018.)

• Servers based on the `socketserver` module, including those defined in `http.server`, `xmlrpc.server` and `wsgiref.simple_server`, now only catch exceptions derived from `Exception`. Therefore if a request handler raises an exception like `SystemExit` or `KeyboardInterrupt`, `handle_error()` is no longer called, and the exception will stop a single-threaded server. (Contributed by Martin Panter in bpo-23430.)

• `spwd.getspnam()` now raises a `PermissionError` instead of `KeyError` if the user doesn’t have privileges.

• The `socket.socket.close()` method now raises an exception if an error (e.g. EBADF) was reported by the underlying system call. (Contributed by Martin Panter in bpo-26685.)

• The `decode_data` argument for the `smtpd.SMTPChannel` and `smtpd.SMTPServer` constructors is now `False` by default. This means that the argument passed to `process_message()` is now a bytes object by default, and `process_message()` will be passed keyword arguments. Code that has already been updated in accordance with the deprecation warning generated by 3.5 will not be affected.
- All optional arguments of the `dump()`, `dumps()`, `load()` and `loads()` functions and `JSONEncoder` and `JSONDecoder` class constructors in the `json` module are now keyword-only. (Contributed by Serhiy Storchaka in bpo-18726.)

- Subclasses of `type` which don’t override `type.__new__` may no longer use the one-argument form to get the type of an object.

- As part of PEP 487, the handling of keyword arguments passed to `type` (other than the metaclass hint, `metaclass`) is now consistently delegated to `object.__init_subclass__()`. This means that `type.__new__()` and `type.__init__()` both now accept arbitrary keyword arguments, but `object.__init_subclass__()` (which is called from `type.__new__()`) will reject them by default. Custom metaclasses accepting additional keyword arguments will need to adjust their calls to `type.__new__()` (whether direct or via `super`) accordingly.

- In `distutils.command.sdist.sdist`, the `default_format` attribute has been removed and is no longer honored. Instead, the gzipped tarfile format is the default on all platforms and no platform-specific selection is made. In environments where distributions are built on Windows and zip distributions are required, configure the project with a `setup.cfg` file containing the following:

```
[sdist]
formats=zip
```

This behavior has also been backported to earlier Python versions by setuptools 26.0.0.
In the `urllib.request` module and the `http.client.HTTPConnection.request()` method, if no Content-Length header field has been specified and the request body is a file object, it is now sent with HTTP 1.1 chunked encoding. If a file object has to be sent to a HTTP 1.0 server, the Content-Length value now has to be specified by the caller. (Contributed by Demian Brecht and Rolf Krahl with tweaks from Martin Panter in bpo-12319.)

- The `DictReader` now returns rows of type `OrderedDict`. (Contributed by Steve Holden in bpo-27842.)

- The `crypt.METHOD_CRYPT` will no longer be added to `crypt.methods` if unsupported by the platform. (Contributed by Victor Stinner in bpo-25287.)

- The `verbose` and `rename` arguments for `namedtuple()` are now keyword-only. (Contributed by Raymond Hettinger in bpo-25628.)

- On Linux, `ctypes.util.find_library()` now looks in `LD_LIBRARY_PATH` for shared libraries. (Contributed by Vinay Sajip in bpo-9998.)

- The `imaplib.IMAP4` class now handles flags containing the `']'` character in messages sent from the server to improve real-world compatibility. (Contributed by Lita Cho in bpo-21815.)

- The `mmap.write()` function now returns the number of bytes written like other write methods. (Contributed by Jakub Stasiak in bpo-26335.)

- The `pkgutil.iter_modules()` and `pkgutil.walk_packages()` functions now return
ModuleInfo named tuples. (Contributed by Ramchandra Apte in bpo-17211.)

- `re.sub()` now raises an error for invalid numerical group references in replacement templates even if the pattern is not found in the string. The error message for invalid group references now includes the group index and the position of the reference. (Contributed by SilentGhost, Serhiy Storchaka in bpo-25953.)

- `zipfile.ZipFile` will now raise `NotImplementedError` for unrecognized compression values. Previously a plain `RuntimeError` was raised. Additionally, calling `ZipFile` methods on a closed `ZipFile` or calling the `write()` method on a `ZipFile` created with mode `['r']` will raise a `ValueError`. Previously, a `RuntimeError` was raised in those scenarios.

- When custom metaclasses are combined with zero-argument `super()` or direct references from methods to the implicit `__class__` closure variable, the implicit `__classcell__` namespace entry must now be passed up to `type.__new__` for initialisation. Failing to do so will result in a `DeprecationWarning` in Python 3.6 and a `RuntimeError` in Python 3.8.

- With the introduction of `ModuleNotFoundError`, import system consumers may start expecting import system replacements to raise that more specific exception when appropriate, rather than the less-specific `ImportError`. To provide future compatibility with such consumers, implementors of alternative import systems that completely replace `__import__()` will need to update their implementations to raise the new subclass when a module can’t
be found at all. Implementors of compliant plugins to the default import system shouldn't need to make any changes, as the default import system will raise the new subclass when appropriate.

Changes in the C API

- The `PyMem_Malloc()` allocator family now uses the `pymalloc` allocator rather than the system `malloc()`. Applications calling `PyMem_Malloc()` without holding the GIL can now crash. Set the `PYTHONMALLOC` environment variable to `debug` to validate the usage of memory allocators in your application. See bpo-26249.
- `Py_Exit()` (and the main interpreter) now override the exit status with 120 if flushing buffered data failed. See bpo-5319.

CPython bytecode changes

There have been several major changes to the bytecode in Python 3.6.

- The Python interpreter now uses a 16-bit wordcode instead of bytecode. (Contributed by Demur Rumed with input and reviews from Serhiy Storchaka and Victor Stinner in bpo-26647 and bpo-28050.)
- The new `FORMAT_VALUE` and `BUILD_STRING` opcodes as part of the formatted string literal implementation. (Contributed by Eric Smith in bpo-25483 and Serhiy Storchaka in bpo-27078.)
- The new `BUILD_CONST_KEY_MAP` opcode to optimize the creation of dictionaries with constant keys. (Contributed by Serhiy Storchaka in bpo-27140.)
- The function call opcodes have been heavily reworked for better
performance and simpler implementation. The MAKE_FUNCTION, CALL_FUNCTION, CALL_FUNCTION_KW and BUILD_MAP_UNPACK_WITH_CALL opcodes have been modified, the new CALL_FUNCTION_EX and BUILD_TUPLE_UNPACK_WITH_CALL have been added, and CALL_FUNCTION_VAR, CALL_FUNCTION_VAR_KW and MAKE_CLOSURE opcodes have been removed. (Contributed by Demur Rumed in bpo-27095, and Serhiy Storchaka in bpo-27213, bpo-28257.)

- The new SETUP_ANNOTATIONS and STORE_ANNOTATION opcodes have been added to support the new variable annotation syntax. (Contributed by Ivan Levkivskyi in bpo-27985.)
Notable changes in Python 3.6.2

New make regen-all build target

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new make regen-all command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see Makefile.pre.in for details.

(Contributed by Victor Stinner in bpo-23404.)

New in version 3.6.2.

Removal of make touch build target

The make touch build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new make regen-all target.

(Contributed by Victor Stinner in bpo-23404.)

Changed in version 3.6.2.
Notable changes in Python 3.6.4

The `PyExc_RecursionErrorInst` singleton that was part of the public API has been removed as its members being never cleared may cause a segfault during finalization of the interpreter. (Contributed by Xavier de Gaye in bpo-22898 and bpo-30697.)
Notable changes in Python 3.6.5

The `locale.localeconv()` function now sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases. (Contributed by Victor Stinner in bpo-31900.)
Notable changes in Python 3.6.7

In 3.6.7 the `tokenize` module now implicitly emits a `NEWLINE` token when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. (Contributed by Ammar Askar in bpo-33899.)
What’s New In Python 3.5

Editors: Elvis Pranskevichus <elvis@magic.io>, Yury Selivanov <yury@magic.io>

This article explains the new features in Python 3.5, compared to 3.4. Python 3.5 was released on September 13, 2015. See the changelog for a full list of changes.

See also: PEP 478 - Python 3.5 Release Schedule
Summary – Release highlights

New syntax features:

- **PEP 492**, coroutines with async and await syntax.
- **PEP 465**, a new matrix multiplication operator: \( a @ b \).
- **PEP 448**, additional unpacking generalizations.

New library modules:

- **typing**: PEP 484 – Type Hints.
- **zipapp**: PEP 441 Improving Python ZIP Application Support.

New built-in features:

- **bytes % args**, **bytearray % args**: PEP 461 – Adding % formatting to bytes and bytearray.
- New **bytes.hex()**, **bytearray.hex()** and **memoryview.hex()** methods. (Contributed by Arnon Yaari in bpo-9951.)
- **memoryview** now supports tuple indexing (including multi-dimensional). (Contributed by Antoine Pitrou in bpo-23632.)
- Generators have a new **gi_yieldfrom** attribute, which returns the object being iterated by **yield from** expressions. (Contributed by Benno Leslie and Yury Selivanov in bpo-24450.)
- A new **RecursionError** exception is now raised when maximum recursion depth is reached. (Contributed by Georg Brandl in bpo-19235.)

CPython implementation improvements:

- When the **LC_TYPE** locale is the POSIX locale (C locale), **sys.stdin** and **sys.stdout** now use the surrogateescape
error handler, instead of the \texttt{strict} error handler. (Contributed by Victor Stinner in \texttt{bpo-19977}.)

- .\texttt{pyo} files are no longer used and have been replaced by a more flexible scheme that includes the optimization level explicitly in .\texttt{pyc} name. (See \texttt{PEP 488 overview}.)
- Built-in and extension modules are now initialized in a multi-phase process, which is similar to how Python modules are loaded. (See \texttt{PEP 489 overview}.)

Significant improvements in the standard library:

- \texttt{collections.OrderedDict} is now \texttt{implemented in C}, which makes it 4 to 100 times faster.
- The \texttt{ssl} module gained \texttt{support for Memory BIO}, which decouples SSL protocol handling from network IO.
- The new \texttt{os.scandir()} function provides a \texttt{better and significantly faster way} of directory traversal.
- \texttt{functools.lru_cache()} has been mostly \texttt{reimplemented in C}, yielding much better performance.
- The new \texttt{subprocess.run()} function provides a \texttt{streamlined way to run subprocesses}.
- The \texttt{traceback} module has been significantly \texttt{enhanced} for improved performance and developer convenience.

Security improvements:

- SSLv3 is now disabled throughout the standard library. It can still be enabled by instantiating a \texttt{ssl.SSLContext} manually. (See \texttt{bpo-22638} for more details; this change was backported to CPython 3.4 and 2.7.)
- HTTP cookie parsing is now stricter, in order to protect against potential injection attacks. (Contributed by Antoine Pitrou in \texttt{bpo-22796}.)
Windows improvements:

- A new installer for Windows has replaced the old MSI. See Using Python on Windows for more information.
- Windows builds now use Microsoft Visual C++ 14.0, and extension modules should use the same.

Please read on for a comprehensive list of user-facing changes, including many other smaller improvements, CPython optimizations, deprecations, and potential porting issues.
New Features

PEP 492 - Coroutines with async and await syntax

**PEP 492** greatly improves support for asynchronous programming in Python by adding **awaitable objects**, **coroutine functions**, **asynchronous iteration**, and **asynchronous context managers**.

Coroutine functions are declared using the new `async def` syntax:

```python
>>> async def coro():
    ...    return 'spam'
```

Inside a coroutine function, the new `await` expression can be used to suspend coroutine execution until the result is available. Any object can be awaited, as long as it implements the `awaitable` protocol by defining the `__await__()` method.

PEP 492 also adds `async for` statement for convenient iteration over asynchronous iterables.

An example of a rudimentary HTTP client written using the new syntax:

```python
import asyncio

async def http_get(domain):
    reader, writer = await asyncio.open_connection(domain)

    writer.write(b'
'.join([
        b'GET / HTTP/1.1',
        b'Host: %b' % domain.encode('latin-1'),
        b'Connection: close',
        b'', b''])
```python
async def coro(name, lock):
    print('coro {}: waiting for lock'.format(name))
    async with lock:
        print('coro {}: holding the lock'.format(name))
        await asyncio.sleep(1)
        print('coro {}: releasing the lock'.format(name))

loop = asyncio.get_event_loop()
lock = asyncio.Lock()
coros = asyncio.gather(coro(1, lock), coro(2, lock))
try:
    loop.run_until_complete(coros)
finally:
    loop.close()
```

Similarly to asynchronous iteration, there is a new syntax for asynchronous context managers. The following script:

```python
import asyncio

async def coro(name, lock):
    print('coro {}: waiting for lock'.format(name))
    async with lock:
        print('coro {}: holding the lock'.format(name))
        await asyncio.sleep(1)
        print('coro {}: releasing the lock'.format(name))

lock = asyncio.Lock()
coros = asyncio.gather(coro(1, lock), coro(2, lock))
try:
    loop.run_until_complete(coros)
finally:
    loop.close()
```
coro 2: waiting for lock
coro 2: holding the lock
coro 1: waiting for lock
coro 2: releasing the lock
coro 1: holding the lock
coro 1: releasing the lock

Note that both async for and async with can only be used inside a coroutine function declared with async def.

Coroutine functions are intended to be run inside a compatible event loop, such as the asyncio loop.

Note:

Changed in version 3.5.2: Starting with CPython 3.5.2, __aiter__ can directly return asynchronous iterators. Returning an awaitable object will result in a PendingDeprecationWarning.

See more details in the Asynchronous Iterators documentation section.

See also:

PEP 492 – Coroutines with async and await syntax
PEP written and implemented by Yury Selivanov.

PEP 465 - A dedicated infix operator for matrix multiplication

PEP 465 adds the @ infix operator for matrix multiplication. Currently, no built-in Python types implement the new operator, however, it can be
implemented by defining `__matmul__()`, `__rmatmul__()`, and `__imatmul__()` for regular, reflected, and in-place matrix multiplication. The semantics of these methods is similar to that of methods defining other infix arithmetic operators.

Matrix multiplication is a notably common operation in many fields of mathematics, science, engineering, and the addition of `@` allows writing cleaner code:

```python
S = (H @ beta - r).T @ inv(H @ V @ H.T) @ (H @ beta)
```

instead of:

```python
S = dot((dot(H, beta) - r).T,
        dot(inv(dot(dot(H, V), H.T)),
             dot(H, beta)) -
```

NumPy 1.10 has support for the new operator:

```python
>>> import numpy

>>> x = numpy.ones(3)
>>> x
array([ 1., 1., 1.])

>>> m = numpy.eye(3)
>>> m
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])

>>> x @ m
array([ 1., 1., 1.])
```
See also:

**PEP 465** – A dedicated infix operator for matrix multiplication

PEP written by Nathaniel J. Smith; implemented by Benjamin Peterson.

**PEP 448** - Additional Unpacking Generalizations

**PEP 448** extends the allowed uses of the * iterable unpacking operator and ** dictionary unpacking operator. It is now possible to use an arbitrary number of unpackings in function calls:

```python
>>> print(*[1], *[2], 3, *[4, 5])
1 2 3 4 5
>>> def fn(a, b, c, d):
...     print(a, b, c, d)
...     ...

>>> fn(**{'a': 1, 'c': 3}, **{'b': 2, 'd': 4})
1 2 3 4
```

Similarly, tuple, list, set, and dictionary displays allow multiple unpackings (see Expression lists and Dictionary displays):

```python
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7}
```
PEP 461 - percent formatting support for bytes and bytearray

PEP 461 adds support for the `%` interpolation operator to `bytes` and `bytearray`.

While interpolation is usually thought of as a string operation, there are cases where interpolation on `bytes` or `bytearrays` makes sense, and the work needed to make up for this missing functionality detracts from the overall readability of the code. This issue is particularly important when dealing with wire format protocols, which are often a mixture of binary and ASCII compatible text.

Examples:

```python
>>> b'Hello %b!' % b'World'
b'Hello World!'

>>> b'x=%i y=%f' % (1, 2.5)
b'x=1 y=2.500000'
```

Unicode is not allowed for `%b`, but it is accepted by `%a` (equivalent of `repr(obj).encode('ascii', 'backslashreplace')`):
>>> b'Hello %b!' % 'World'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %b requires bytes, or an object that implements __bytes__, not 'str'

>>> b'price: %a' % '10€'
b'price: '10\u20ac'"

Note that %s and %r conversion types, although supported, should only be used in codebases that need compatibility with Python 2.

See also:

**PEP 461** – Adding % formatting to bytes and bytearray
   PEP written by Ethan Furman; implemented by Neil Schemenauer and Ethan Furman.

**PEP 484** - Type Hints

Function annotation syntax has been a Python feature since version 3.0 (**PEP 3107**), however the semantics of annotations has been left undefined.

Experience has shown that the majority of function annotation uses were to provide type hints to function parameters and return values. It became evident that it would be beneficial for Python users, if the standard library included the base definitions and tools for type annotations.

**PEP 484** introduces a **provisional module** to provide these standard definitions and tools, along with some conventions for situations where annotations are not available.
For example, here is a simple function whose argument and return type are declared in the annotations:

```python
def greeting(name: str) -> str:
    return 'Hello ' + name
```

While these annotations are available at runtime through the usual `__annotations__` attribute, *no automatic type checking happens at runtime*. Instead, it is assumed that a separate off-line type checker (e.g. `mypy`) will be used for on-demand source code analysis.

The type system supports unions, generic types, and a special type named `Any` which is consistent with (i.e. assignable to and from) all types.

See also:

- `typing` module documentation
- **PEP 484 – Type Hints**
  PEP written by Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa; implemented by Guido van Rossum.
- **PEP 483 – The Theory of Type Hints**
  PEP written by Guido van Rossum

**PEP 471 - os.scandir() function – a better and faster directory iterator**

**PEP 471** adds a new directory iteration function, `os.scandir()`, to the standard library. Additionally, `os.walk()` is now implemented using `scandir`, which makes it 3 to 5 times faster on POSIX systems and 7 to 20 times faster on Windows systems. This is largely achieved
by greatly reducing the number of calls to `os.stat()` required to walk a directory tree.

Additionally, `scandir` returns an iterator, as opposed to returning a list of file names, which improves memory efficiency when iterating over very large directories.

The following example shows a simple use of `os.scandir()` to display all the files (excluding directories) in the given `path` that don't start with `.'`. The `entry.is_file()` call will generally not make an additional system call:

```python
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

See also:

**PEP 471** – `os.scandir()` function – a better and faster directory iterator
   PEP written and implemented by Ben Hoyt with the help of Victor Stinner.

**PEP 475**: Retry system calls failing with EINTR

An `errno.EINTR` error code is returned whenever a system call, that is waiting for I/O, is interrupted by a signal. Previously, Python would raise `InterruptedError` in such cases. This meant that, when writing a Python application, the developer had two choices:

1. Ignore the `InterruptedError`.
2. Handle the `InterruptedError` and attempt to restart the
interrupted system call at every call site.

The first option makes an application fail intermittently. The second option adds a large amount of boilerplate that makes the code nearly unreadable. Compare:

```python
print("Hello World")
```

and:

```python
while True:
    try:
        print("Hello World")
        break
    except InterruptedError:
        continue
```

**PEP 475** implements automatic retry of system calls on `EINTR`. This removes the burden of dealing with `EINTR` or `InterruptedError` in user code in most situations and makes Python programs, including the standard library, more robust. Note that the system call is only retried if the signal handler does not raise an exception.

Below is a list of functions which are now retried when interrupted by a signal:

- `open()` and `io.open();`
- functions of the `faulthandler` module;
- `os` functions: `fchdir()`, `fchmod()`, `fchown()`, `fdatasync()`, `fstat()`, `fstatvfs()`, `fsync()`, `ftruncate()`, `mkfifo()`, `mknod()`, `open()`, `posix_fadvise()`, `posix_fallocate()`, `pread()`, `pwrite()`, `read()`, `readv()`, `sendfile()`, `wait3()`, `wait4()`, `wait()`, `waitid()`, `waitpid()`, `write()`, `writev()`.
writev();
- special cases: os.close() and os.dup2() now ignore EINTR errors; the syscall is not retried (see the PEP for the rationale);
- select functions: devpoll.poll(), epoll.poll(), kqueue.control(), poll.poll(), select();
- methods of the socket class: accept(), connect() (except for non-blocking sockets), recv(), recvfrom(), recvmsg(), send(), sendall(), sendmsg(), sendto();
- signal.sigtimedwait() and signal.sigwaitinfo();
- time.sleep().

See also:

**PEP 475** – Retry system calls failing with EINTR

PEP and implementation written by Charles-François Natali and Victor Stinner, with the help of Antoine Pitrou (the French connection).

**PEP 479**: Change StopIteration handling inside generators

The interaction of generators and StopIteration in Python 3.4 and earlier was sometimes surprising, and could conceal obscure bugs. Previously, StopIteration raised accidentally inside a generator function was interpreted as the end of the iteration by the loop construct driving the generator.

**PEP 479** changes the behavior of generators: when a StopIteration exception is raised inside a generator, it is replaced with a RuntimeError before it exits the generator frame. The main goal of this change is to ease debugging in the situation where an unguarded next() call raises StopIteration and causes the
iteration controlled by the generator to terminate silently. This is particularly pernicious in combination with the `yield` from construct.

This is a backwards incompatible change, so to enable the new behavior, a `__future__` import is necessary:

```python
>>> from __future__ import generator_stop

>>> def gen():
...     next(iter([]))
...     yield
...
>>> next(gen())
Traceback (most recent call last):
  File "<stdin>", line 2, in gen
StopIteration

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator raised StopIteration

Without a `__future__` import, a `PendingDeprecationWarning` will be raised whenever a `StopIteration` exception is raised inside a generator.

See also:

**PEP 479 – Change StopIteration handling inside generators**

PEP written by Chris Angelico and Guido van Rossum. Implemented by Chris Angelico, Yury Selivanov and Nick Coghlan.
PEP 485: A function for testing approximate equality

PEP 485 adds the `math.isclose()` and `cmath.isclose()` functions which tell whether two values are approximately equal or “close” to each other. Whether or not two values are considered close is determined according to given absolute and relative tolerances. Relative tolerance is the maximum allowed difference between `isclose` arguments, relative to the larger absolute value:

```python
>>> import math
>>> a = 5.0
>>> b = 4.99998
>>> math.isclose(a, b, rel_tol=1e-5)
True
>>> math.isclose(a, b, rel_tol=1e-6)
False
```

It is also possible to compare two values using absolute tolerance, which must be a non-negative value:

```python
>>> import math
>>> a = 5.0
>>> b = 4.99998
>>> math.isclose(a, b, abs_tol=0.00003)
True
>>> math.isclose(a, b, abs_tol=0.00001)
False
```

See also:

PEP 485 – A function for testing approximate equality

PEP written by Christopher Barker; implemented by Chris Barker
PEP 486: Make the Python Launcher aware of virtual environments

**PEP 486** makes the Windows launcher (see **PEP 397**) aware of an active virtual environment. When the default interpreter would be used and the VIRTUAL_ENV environment variable is set, the interpreter in the virtual environment will be used.

**See also:**

**PEP 486** – Make the Python Launcher aware of virtual environments

PEP written and implemented by Paul Moore.

PEP 488: Elimination of PYO files

**PEP 488** does away with the concept of .pyo files. This means that .pyc files represent both unoptimized and optimized bytecode. To prevent the need to constantly regenerate bytecode files, .pyc files now have an optional `opt-` tag in their name when the bytecode is optimized. This has the side-effect of no more bytecode file name clashes when running under either `-O` or `-OO`. Consequently, bytecode files generated from `-O`, and `-OO` may now exist simultaneously. `importlib.util.cache_from_source()` has an updated API to help with this change.

**See also:**

**PEP 488** – Elimination of PYO files
PEP 489: Multi-phase extension module initialization

PEP 489 updates extension module initialization to take advantage of the two step module loading mechanism introduced by PEP 451 in Python 3.4.

This change brings the import semantics of extension modules that opt-in to using the new mechanism much closer to those of Python source and bytecode modules, including the ability to use any valid identifier as a module name, rather than being restricted to ASCII.

See also:

PEP 489 – Multi-phase extension module initialization
  PEP written by Petr Viktorin, Stefan Behnel, and Nick Coghlan; implemented by Petr Viktorin.
Other Language Changes

Some smaller changes made to the core Python language are:

- Added the "namereplace" error handlers. The "backslashreplace" error handlers now work with decoding and translating. (Contributed by Serhiy Storchaka in bpo-19676 and bpo-22286.)
- The -b option now affects comparisons of bytes with int. (Contributed by Serhiy Storchaka in bpo-23681.)
- New Kazakh kz1048 and Tajik koi8_t codecs. (Contributed by Serhiy Storchaka in bpo-22682 and bpo-22681.)
- Property docstrings are now writable. This is especially useful for collections.namedtuple() docstrings. (Contributed by Berker Peksag in bpo-24064.)
- Circular imports involving relative imports are now supported. (Contributed by Brett Cannon and Antoine Pitrou in bpo-17636.)
New Modules

typing

The new typing provisional module provides standard definitions and tools for function type annotations. See Type Hints for more information.

zipapp

The new zipapp module (specified in PEP 441) provides an API and command line tool for creating executable Python Zip Applications, which were introduced in Python 2.6 in bpo-1739468, but which were not well publicized, either at the time or since.

With the new module, bundling your application is as simple as putting all the files, including a __main__.py file, into a directory myapp and running:

```
$ python -m zipapp myapp
$ python myapp.pyz
```

The module implementation has been contributed by Paul Moore in bpo-23491.

See also: PEP 441 – Improving Python ZIP Application Support
Improved Modules

argparse

The `ArgumentParser` class now allows disabling abbreviated usage of long options by setting `allow_abbrev` to `False`. (Contributed by Jonathan Paugh, Steven Bethard, paul j3 and Daniel Eriksson in bpo-14910.)

asyncio

Since the `asyncio` module is provisional, all changes introduced in Python 3.5 have also been backported to Python 3.4.x.

Notable changes in the `asyncio` module since Python 3.4.0:

- New debugging APIs: `loop.set_debug()` and `loop.get_debug()` methods. (Contributed by Victor Stinner.)
- The proactor event loop now supports SSL. (Contributed by Antoine Pitrou and Victor Stinner in bpo-22560.)
- A new `loop.is_closed()` method to check if the event loop is closed. (Contributed by Victor Stinner in bpo-21326.)
- A new `loop.create_task()` to conveniently create and schedule a new `Task` for a coroutine. The `create_task` method is also used by all asyncio functions that wrap coroutines into tasks, such as `asyncio.wait()`, `asyncio.gather()`, etc. (Contributed by Victor Stinner.)
- A new `transport.get_write_buffer_limits()` method to inquire for high- and low- water limits of the flow control. (Contributed by Victor Stinner.)
- The `async()` function is deprecated in favor of `ensure_future()`. (Contributed by Yury Selivanov.)
New `loop.set_task_factory()` and `loop.get_task_factory()` methods to customize the task factory that `loop.create_task()` method uses. (Contributed by Yury Selivanov.)

New `Queue.join()` and `Queue.task_done()` queue methods. (Contributed by Victor Stinner.)

The `JoinableQueue` class was removed, in favor of the `asyncio.Queue` class. (Contributed by Victor Stinner.)

Updates in 3.5.1:

- The `ensure_future()` function and all functions that use it, such as `loop.run_until_complete()`, now accept all kinds of `awaitable` objects. (Contributed by Yury Selivanov.)
- New `run_coroutine_threadsafe()` function to submit coroutines to event loops from other threads. (Contributed by Vincent Michel.)
- New `Transport.is_closing()` method to check if the transport is closing or closed. (Contributed by Yury Selivanov.)
- The `loop.create_server()` method can now accept a list of hosts. (Contributed by Yann Sionneau.)

Updates in 3.5.2:

- New `loop.create_future()` method to create `Future` objects. This allows alternative event loop implementations, such as `uvloop`, to provide a faster `asyncio.Future` implementation. (Contributed by Yury Selivanov.)
- New `loop.get_exception_handler()` method to get the current exception handler. (Contributed by Yury Selivanov.)
- New `StreamReader.readuntil()` method to read data from the stream until a separator bytes sequence appears. (Contributed by Mark Korenberg.)
- The `loop.create_connection()` and
loop.create_server() methods are optimized to avoid calling the system getaddrinfo function if the address is already resolved. (Contributed by A. Jesse Jiryu Davis.)

- The loop.sock_connect(sock, address) no longer requires the address to be resolved prior to the call. (Contributed by A. Jesse Jiryu Davis.)

bz2

The BZ2Decompressor.decompress method now accepts an optional max_length argument to limit the maximum size of decompressed data. (Contributed by Nikolaus Rath in bpo-15955.)

cgi

The FieldStorage class now supports the context manager protocol. (Contributed by Berker Peksag in bpo-20289.)

cmath

A new function isclose() provides a way to test for approximate equality. (Contributed by Chris Barker and Tal Einat in bpo-24270.)

code

The InteractiveInterpreter.showtraceback() method now prints the full chained traceback, just like the interactive interpreter. (Contributed by Claudiu Popa in bpo-17442.)

collections
The `OrderedDict` class is now implemented in C, which makes it 4 to 100 times faster. (Contributed by Eric Snow in `bpo-16991`.)

`OrderedDict.items()`, `OrderedDict.keys()`, `OrderedDict.values()` views now support `reversed()` iteration. (Contributed by Serhiy Storchaka in `bpo-19505`.)

The `deque` class now defines `index()`, `insert()`, and `copy()`, and supports the `+` and `*` operators. This allows deques to be recognized as a `MutableSequence` and improves their substitutability for lists. (Contributed by Raymond Hettinger in `bpo-23704`.)

Docstrings produced by `namedtuple()` can now be updated:

```python
Point = namedtuple('Point', ['x', 'y'])
Point.__doc__ += ': Cartesian coordinate'
Point.x.__doc__ = 'abscissa'
Point.y.__doc__ = 'ordinate'
```

(Contributed by Berker Peksag in `bpo-24064`.)

The `UserString` class now implements the `__getnewargs__()`, `__rmod__()`, `casefold()`, `format_map()`, `isprintable()`, and `maketrans()` methods to match the corresponding methods of `str`. (Contributed by Joe Jevnik in `bpo-22189`.)

**collections.abc**

The `Sequence.index()` method now accepts `start` and `stop` arguments to match the corresponding methods of `tuple`, `list`, etc. (Contributed by Devin Jeanpierre in `bpo-23086`.)
A new **Generator** abstract base class. (Contributed by Stefan Behnel in *bpo-24018*.)

New **Awaitable**, **Coroutine**, **AsyncIterator**, and **AsyncIterable** abstract base classes. (Contributed by Yury Selivanov in *bpo-24184*.)

For earlier Python versions, a backport of the new ABCs is available in an external **PyPI package**.

**compileall**

A new **compileall** option, `-j N`, allows running $N$ workers simultaneously to perform parallel bytecode compilation. The **compile_dir()** function has a corresponding **workers** parameter. (Contributed by Claudiu Popa in *bpo-16104*.)

Another new option, `-r`, allows controlling the maximum recursion level for subdirectories. (Contributed by Claudiu Popa in *bpo-19628*.)

The `-q` command line option can now be specified more than once, in which case all output, including errors, will be suppressed. The corresponding **quiet** parameter in **compile_dir()**, **compile_file()**, and **compile_path()** can now accept an integer value indicating the level of output suppression. (Contributed by Thomas Kluyver in *bpo-21338*.)

**concurrent.futures**

The **Executor.map()** method now accepts a **chunksize** argument to allow batching of tasks to improve performance when **ProcessPoolExecutor()** is used. (Contributed by Dan O'Reilly in
The number of workers in the `ThreadPoolExecutor` constructor is optional now. The default value is 5 times the number of CPUs. (Contributed by Claudiu Popa in bpo-21527.)

`configparser`

`configparser` now provides a way to customize the conversion of values by specifying a dictionary of converters in the `ConfigParser` constructor, or by defining them as methods in `ConfigParser` subclasses. Converters defined in a parser instance are inherited by its section proxies.

Example:

```python
>>> import configparser
>>> conv = {}
>>> conv['list'] = lambda v: [e.strip() for e in v.split()]
>>> cfg = configparser.ConfigParser(converters=conv)
>>> cfg.read_string("...
... [s]
... list = a b c d e f g
... "")
>>> cfg.get('s', 'list')
'a b c d e f g'
>>> cfg.getlist('s', 'list')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> section = cfg['s']
>>> section.getlist('list')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

(Contributed by Łukasz Langa in bpo-18159.)
contextlib

The new `redirect_stderr()` context manager (similar to `redirect_stdout()`) makes it easier for utility scripts to handle inflexible APIs that write their output to `sys.stderr` and don't provide any options to redirect it:

```python
>>> import contextlib, io, logging
>>> f = io.StringIO()
>>> with contextlib.redirect_stderr(f):
...     logging.warning('warning')
...     ...
>>> f.getvalue()
'WARNING:root:warning\n'
```

(Contributed by Berker Peksag in bpo-22389.)

CSV

The `writerow()` method now supports arbitrary iterables, not just sequences. (Contributed by Serhiy Storchaka in bpo-23171.)

curses

The new `update_lines_cols()` function updates the LINES and COLS environment variables. This is useful for detecting manual screen resizing. (Contributed by Arnon Yaari in bpo-4254.)

dbm

dumb.open always creates a new database when the flag has the
difflib

The charset of HTML documents generated by `HtmlDiff.make_file()` can now be customized by using a new `charset` keyword-only argument. The default charset of HTML document changed from "ISO-8859-1" to "utf-8". (Contributed by Berker Peksag in bpo-2052.)

The `diff_bytes()` function can now compare lists of byte strings. This fixes a regression from Python 2. (Contributed by Terry J. Reedy and Greg Ward in bpo-17445.)

distutils

Both the `build` and `build_ext` commands now accept a `-j` option to enable parallel building of extension modules. (Contributed by Antoine Pitrou in bpo-5309.)

The `distutils` module now supports xz compression, and can be enabled by passing `xztar` as an argument to `bdist --format`. (Contributed by Serhiy Storchaka in bpo-16314.)

doctest

The `DocTestSuite()` function returns an empty `unittest.TestSuite` if `module` contains no docstrings, instead of raising `ValueError`. (Contributed by Glenn Jones in bpo-15916.)

email
A new policy option `Policy.mangle_from_` controls whether or not lines that start with "From " in email bodies are prefixed with a "">" character by generators. The default is `True` for `compat32` and `False` for all other policies. (Contributed by Milan Oberkirch in bpo-20098.)

A new `Message.get_content_disposition()` method provides easy access to a canonical value for the `Content-Disposition` header. (Contributed by Abhilash Raj in bpo-21083.)

A new policy option `EmailPolicy.utf8` can be set to `True` to encode email headers using the UTF-8 charset instead of using encoded words. This allows `Messages` to be formatted according to RFC 6532 and used with an SMTP server that supports the RFC 6531 SMTPUTF8 extension. (Contributed by R. David Murray in bpo-24211.)

The `mime.text.MIMEText` constructor now accepts a `charset.Charset` instance. (Contributed by Claude Paroz and Berker Peksag in bpo-16324.)

**enum**

The `Enum` callable has a new parameter `start` to specify the initial number of enum values if only `names` are provided:

```python
>>> Animal = enum.Enum('Animal', 'cat dog', start=10)
>>> Animal.cat
<Animal.cat: 10>
>>> Animal.dog
<Animal.dog: 11>
```
faulthandler

The `enable()`, `register()`, `dump_traceback()` and `dump_traceback_later()` functions now accept file descriptors in addition to file-like objects. (Contributed by Wei Wu in bpo-23566.)

functools

Most of the `lru_cache()` machinery is now implemented in C, making it significantly faster. (Contributed by Matt Joiner, Alexey Kachayev, and Serhiy Storchaka in bpo-14373.)

glob

The `iglob()` and `glob()` functions now support recursive search in subdirectories, using the `***` pattern. (Contributed by Serhiy Storchaka in bpo-13968.)

gzip

The `mode` argument of the `GzipFile` constructor now accepts "x" to request exclusive creation. (Contributed by Tim Heaney in bpo-19222.)

heapq

Element comparison in `merge()` can now be customized by passing a `key function` in a new optional `key` keyword argument, and a new optional `reverse` keyword argument can be used to reverse element
comparison:

```python
>>> import heapq
>>> a = ['9', '777', '55555']
>>> b = ['88', '6666']
>>> list(heapq.merge(a, b, key=len))
['9', '88', '777', '6666', '55555']
>>> list(heapq.merge(reversed(a), reversed(b), key=len))
['55555', '6666', '777', '88', '9']
```

(Contributed by Raymond Hettinger in bpo-13742.)

http

A new HTTPStatus enum that defines a set of HTTP status codes, reason phrases and long descriptions written in English. (Contributed by Demian Brecht in bpo-21793.)

http.client

HTTPConnection.getresponse() now raises a RemoteDisconnected exception when a remote server connection is closed unexpectedly. Additionally, if a ConnectionError (of which RemoteDisconnected is a subclass) is raised, the client socket is now closed automatically, and will reconnect on the next request:

```python
import http.client
conn = http.client.HTTPConnection('www.python.org')
for retries in range(3):
    try:
        conn.request('GET', '/')
        resp = conn.getresponse()
    except http.client.RemoteDisconnected:
```
(Contributed by Martin Panter in bpo-3566.)

idlelib and IDLE

Since idlelib implements the IDLE shell and editor and is not intended for import by other programs, it gets improvements with every release. See Lib/idlelib/NEWS.txt for a cumulative list of changes since 3.4.0, as well as changes made in future 3.5.x releases. This file is also available from the IDLE Help ▶ About IDLE dialog.

imaplib

The **IMAP4** class now supports the context manager protocol. When used in a `with` statement, the IMAP4 **LOGOUT** command will be called automatically at the end of the block. (Contributed by Tarek Ziadé and Serhiy Storchaka in bpo-4972.)

The `imaplib` module now supports **RFC 5161** (ENABLE Extension) and **RFC 6855** (UTF-8 Support) via the `IMAP4.enable()` method. A new `IMAP4.utf8_enabled` attribute tracks whether or not **RFC 6855** support is enabled. (Contributed by Milan Oberkirch, R. David Murray, and Maciej Szulik in bpo-21800.)

The `imaplib` module now automatically encodes non-ASCII string usernames and passwords using UTF-8, as recommended by the RFCs. (Contributed by Milan Oberkirch in bpo-21800.)

imghdr
The `what()` function now recognizes the OpenEXR format (contributed by Martin Vignali and Claudiu Popa in bpo-20295), and the WebP format (contributed by Fabrice Aneche and Claudiu Popa in bpo-20197.)

**importlib**

The `util.LazyLoader` class allows for lazy loading of modules in applications where startup time is important. (Contributed by Brett Cannon in bpo-17621.)

The `abc.InspectLoader.source_to_code()` method is now a static method. This makes it easier to initialize a module object with code compiled from a string by running `exec(code, module.__dict__)`. (Contributed by Brett Cannon in bpo-21156.)

The new `util.module_from_spec()` function is now the preferred way to create a new module. As opposed to creating a `types.ModuleType` instance directly, this new function will set the various import-controlled attributes based on the passed-in spec object. (Contributed by Brett Cannon in bpo-20383.)

**inspect**

Both the `Signature` and `Parameter` classes are now picklable and hashable. (Contributed by Yury Selivanov in bpo-20726 and bpo-20334.)

A new `BoundArguments.apply_defaults()` method provides a way to set default values for missing arguments:

```python
>>> def foo(a, b='ham', *args): pass
```
A new class method `Signature.from_callable()` makes subclassing of `Signature` easier. (Contributed by Yury Selivanov and Eric Snow in `bpo-17373`.)

The `signature()` function now accepts a `follow_wrapped` optional keyword argument, which, when set to `False`, disables automatic following of `__wrapped__` links. (Contributed by Yury Selivanov in `bpo-20691`.)

A set of new functions to inspect coroutine functions and coroutine objects has been added: `iscoroutine()`, `iscoroutinefunction()`, `isawaitable()`, `getcoroutinelocals()`, and `getcoroutinestate()`. (Contributed by Yury Selivanov in `bpo-24017` and `bpo-24400`.)

The `stack()`, `trace()`, `getouterframes()`, and `getinnerframes()` functions now return a list of named tuples. (Contributed by Daniel Shahaf in `bpo-16808`.)

io

A new `BufferedIOBase.readinto1()` method, that uses at most one call to the underlying raw stream’s `RawIOBase.read()` or `RawIOBase.readinto()` methods. (Contributed by Nikolaus Rath in `bpo-20578`.)
ipaddress

Both the IPv4Network and IPv6Network classes now accept an (address, netmask) tuple argument, so as to easily construct network objects from existing addresses:

```python
>>> import ipaddress
>>> ipaddress.IPv4Network(('127.0.0.0', 8))
IPv4Network('127.0.0.0/8')
>>> ipaddress.IPv4Network(('127.0.0.0', '255.0.0.0'))
IPv4Network('127.0.0.0/8')
```

(Contributed by Peter Moody and Antoine Pitrou in bpo-16531.)

A new reverse_pointer attribute for the IPv4Network and IPv6Network classes returns the name of the reverse DNS PTR record:

```python
>>> import ipaddress
>>> addr = ipaddress.IPv4Address('127.0.0.1')
>>> addr.reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> addr6 = ipaddress.IPv6Address('::1')
>>> addr6.reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa'
```

(Contributed by Leon Weber in bpo-20480.)

json

The json.tool command line interface now preserves the order of keys in JSON objects passed in input. The new --sort-keys option
can be used to sort the keys alphabetically. (Contributed by Berker Peksag in bpo-21650.)

JSON decoder now raises JSONDecodeError instead of ValueError to provide better context information about the error. (Contributed by Serhiy Storchaka in bpo-19361.)

**linecache**

A new `lazycache()` function can be used to capture information about a non-file-based module to permit getting its lines later via `getline()`. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely. (Contributed by Robert Collins in bpo-17911.)

**locale**

A new `delocalize()` function can be used to convert a string into a normalized number string, taking the `LC_NUMERIC` settings into account:

```python
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, 'de_DE.UTF-8'
'de_DE.UTF-8'
'1.234,56')
'1234.56'
>>> locale.setlocale(locale.LC_NUMERIC, 'en_US.UTF-8'
'en_US.UTF-8'
'1,234.56')
'1234.56'
```

(Contributed by Cédric Krier in bpo-13918.)
logging

All logging methods (Logger log(), exception(), critical(), debug(), etc.), now accept exception instances as an exc_info argument, in addition to boolean values and exception tuples:

```python
>>> import logging
>>> try:
...     1/0
... except ZeroDivisionError as ex:
...     logging.error('exception', exc_info=ex)
ERROR:root:exception
```

(Contributed by Yury Selivanov in bpo-20537.)

The handlers.HTTPHandler class now accepts an optional ssl.SSLContext instance to configure SSL settings used in an HTTP connection. (Contributed by Alex Gaynor in bpo-22788.)

The handlers.QueueListener class now takes a respect_handler_level keyword argument which, if set to True, will pass messages to handlers taking handler levels into account. (Contributed by Vinay Sajip.)

lzma

The LZMADecompressor.decompress() method now accepts an optional max_length argument to limit the maximum size of decompressed data. (Contributed by Martin Panter in bpo-15955.)

math
Two new constants have been added to the `math` module: `inf` and `nan`. (Contributed by Mark Dickinson in bpo-23185.)

A new function `isclose()` provides a way to test for approximate equality. (Contributed by Chris Barker and Tal Einat in bpo-24270.)

A new `gcd()` function has been added. The `fractions.gcd()` function is now deprecated. (Contributed by Mark Dickinson and Serhiy Storchaka in bpo-22486.)

**multiprocessing**

`sharedctypes.synchronized()` objects now support the `context manager` protocol. (Contributed by Charles-François Natali in bpo-21565.)

**operator**

`attrgetter()`, `itemgetter()`, and `methodcaller()` objects now support pickling. (Contributed by Josh Rosenberg and Serhiy Storchaka in bpo-22955.)

New `matmul()` and `imatmul()` functions to perform matrix multiplication. (Contributed by Benjamin Peterson in bpo-21176.)

**OS**

The new `scandir()` function returning an iterator of `DirEntry` objects has been added. If possible, `scandir()` extracts file attributes while scanning a directory, removing the need to perform subsequent system calls to determine file type or attributes, which may
significantly improve performance. (Contributed by Ben Hoyt with the help of Victor Stinner in \texttt{bpo-22524}.)

On Windows, a new \texttt{stat\_result.st\_file\_attributes} attribute is now available. It corresponds to the \texttt{dwFileAttributes} member of the \texttt{BY\_HANDLE\_FILE\_INFORMATION} structure returned by \texttt{GetFileInformationByHandle()}. (Contributed by Ben Hoyt in \texttt{bpo-21719}.)

The \texttt{urandom()} function now uses the \texttt{getrandom()} syscall on Linux 3.17 or newer, and \texttt{getentropy()} on OpenBSD 5.6 and newer, removing the need to use \texttt{/dev/urandom} and avoiding failures due to potential file descriptor exhaustion. (Contributed by Victor Stinner in \texttt{bpo-22181}.)

New \texttt{get\_blocking()} and \texttt{set\_blocking()} functions allow getting and setting a file descriptor's blocking mode (\texttt{O\_NONBLOCK}). (Contributed by Victor Stinner in \texttt{bpo-22054}.)

The \texttt{truncate()} and \texttt{ftruncate()} functions are now supported on Windows. (Contributed by Steve Dower in \texttt{bpo-23668}.)

There is a new \texttt{os.path.commonpath()} function returning the longest common sub-path of each passed pathname. Unlike the \texttt{os.path.commonprefix()} function, it always returns a valid path:

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib', '/usr'])
'/usr'
```

(Contributed by Rafik Draoui and Serhiy Storchaka in \texttt{bpo-10395}.)

pathlib

The new `Path.samefile()` method can be used to check whether the path points to the same file as another path, which can be either another `Path` object, or a string:

```python
>>> import pathlib
>>> p1 = pathlib.Path('/etc/hosts')
>>> p2 = pathlib.Path('/etc/../etc/hosts')
>>> p1.samefile(p2)
True
```

(Contributed by Vajrasky Kok and Antoine Pitrou in bpo-19775.)

The `Path.mkdir()` method now accepts a new optional `exist_ok` argument to match `mkdir -p` and `os.makedirs()` functionality. (Contributed by Berker Peksag in bpo-21539.)

There is a new `Path.expanduser()` method to expand `~` and `~user` prefixes. (Contributed by Serhiy Storchaka and Claudiu Popa in bpo-19776.)

A new `Path.home()` class method can be used to get a `Path` instance representing the user’s home directory. (Contributed by Victor Salgado and Mayank Tripathi in bpo-19777.)

New `Path.write_text()`, `Path.read_text()`, `Path.write_bytes()`, `Path.read_bytes()` methods to simplify read/write operations on files.

The following code snippet will create or rewrite existing file `~/spam42`: 
```python
>>> import pathlib
>>> p = pathlib.Path('~/spam42')
>>> p.expanduser().write_text('ham')
3

(Contributed by Christopher Welborn in bpo-20218.)

pickle

Nested objects, such as unbound methods or nested classes, can now be pickled using pickle protocols older than protocol version 4. Protocol version 4 already supports these cases. (Contributed by Serhiy Storchaka in bpo-23611.)

poplib

A new POP3.utf8() command enables RFC 6856 (Internationalized Email) support, if a POP server supports it. (Contributed by Milan OberKirch in bpo-21804.)

re

References and conditional references to groups with fixed length are now allowed in lookahead assertions:

```
The number of capturing groups in regular expressions is no longer limited to 100. (Contributed by Serhiy Storchaka in bpo-22437.)

The `sub()` and `subn()` functions now replace unmatched groups with empty strings instead of raising an exception. (Contributed by Serhiy Storchaka in bpo-1519638.)

The `re.error` exceptions have new attributes, `msg`, `pattern`, `pos`, `lineno`, and `colno`, that provide better context information about the error:

```python
>>> re.compile("""
... (?x)
... .++
... """")
Traceback (most recent call last):
...
sre_constants.error: multiple repeat at position 16
```

(readline)

A new `append_history_file()` function can be used to append the specified number of trailing elements in history to the given file. (Contributed by Bruno Cauet in bpo-22940.)

(selectors)

The new `DevpollSelector` supports efficient `/dev/poll` polling
on Solaris. (Contributed by Giampaolo Rodola’ in bpo-18931.)

**shutil**

The `move()` function now accepts a `copy_function` argument, allowing, for example, the `copy()` function to be used instead of the default `copy2()` if there is a need to ignore file metadata when moving. (Contributed by Claudiu Popa in bpo-19840.)

The `make_archive()` function now supports the `xztar` format. (Contributed by Serhiy Storchaka in bpo-5411.)

**signal**

On Windows, the `set_wakeup_fd()` function now also supports socket handles. (Contributed by Victor Stinner in bpo-22018.)

Various SIG* constants in the `signal` module have been converted into Enums. This allows meaningful names to be printed during debugging, instead of integer “magic numbers”. (Contributed by Giampaolo Rodola’ in bpo-21076.)

**smtpd**

Both the `SMTPServer` and `SMTPChannel` classes now accept a `decode_data` keyword argument to determine if the DATA portion of the SMTP transaction is decoded using the "utf-8" codec or is instead provided to the `SMTPServer.process_message()` method as a byte string. The default is True for backward compatibility reasons, but will change to False in Python 3.6. If `decode_data` is set to False, the `process_message` method must be prepared to
accept keyword arguments. (Contributed by Maciej Szulik in bpo-19662.)

The SMTPServer class now advertises the 8BITMIME extension (RFC 6152) if decode_data has been set True. If the client specifies BODY=8BITMIME on the MAIL command, it is passed to SMTPServer.process_message() via the mail_options keyword. (Contributed by Milan Oberkirch and R. David Murray in bpo-21795.)

The SMTPServer class now also supports the SMTPUTF8 extension (RFC 6531: Internationalized Email). If the client specified SMTPUTF8 BODY=8BITMIME on the MAIL command, they are passed to SMTPServer.process_message() via the mail_options keyword. It is the responsibility of the process_message method to correctly handle the SMTPUTF8 data. (Contributed by Milan Oberkirch in bpo-21725.)

It is now possible to provide, directly or via name resolution, IPv6 addresses in the SMTPServer constructor, and have it successfully connect. (Contributed by Milan Oberkirch in bpo-14758.)

smtplib

A new SMTP.auth() method provides a convenient way to implement custom authentication mechanisms. (Contributed by Milan Oberkirch in bpo-15014.)

The SMTP.set_debuglevel() method now accepts an additional debuglevel (2), which enables timestamps in debug messages. (Contributed by Gavin Chappell and Maciej Szulik in bpo-16914.)

Both the SMTP.sendmail() and SMTP.send_message() methods
now support RFC 6531 (SMTPUTF8). (Contributed by Milan Oberkirch and R. David Murray in bpo-22027.)

sndhdr

The what() and whathdr() functions now return a namedtuple(). (Contributed by Claudiu Popa in bpo-18615.)

socket

Functions with timeouts now use a monotonic clock, instead of a system clock. (Contributed by Víctor Stinner in bpo-22043.)

A new socket.sendfile() method allows sending a file over a socket by using the high-performance os.sendfile() function on UNIX, resulting in uploads being from 2 to 3 times faster than when using plain socket.send(). (Contributed by Giampaolo Rodola’ in bpo-17552.)

The socket.sendall() method no longer resets the socket timeout every time bytes are received or sent. The socket timeout is now the maximum total duration to send all data. (Contributed by Victor Stinner in bpo-23853.)

The backlog argument of the socket.listen() method is now optional. By default it is set to SOMAXCONN or to 128, whichever is less. (Contributed by Charles-François Natali in bpo-21455.)

ssl

Memory BIO Support
The new **SSLObject** class has been added to provide SSL protocol support for cases when the network I/O capabilities of **SSLSocket** are not necessary or are suboptimal. **SSLObject** represents an SSL protocol instance, but does not implement any network I/O methods, and instead provides a memory buffer interface. The new **MemoryBIO** class can be used to pass data between Python and an SSL protocol instance.

The memory BIO SSL support is primarily intended to be used in frameworks implementing asynchronous I/O for which **SSLSocket**'s readiness model (“select/poll”) is inefficient.

A new **SSLContext.wrap_bio()** method can be used to create a new **SSLObject** instance.

**Application-Layer Protocol Negotiation Support**

Where OpenSSL support is present, the **ssl** module now implements the Application-Layer Protocol Negotiation TLS extension as described in RFC 7301.

The new **SSLContext.set_alpn_protocols()** can be used to specify which protocols a socket should advertise during the TLS handshake.

The new **SSLSocket.selected_alpn_protocol()** returns the protocol that was selected during the TLS handshake. The **HAS_ALPN** flag indicates whether ALPN support is present.
Other Changes

There is a new `SSLSocket.version()` method to query the actual protocol version in use. (Contributed by Antoine Pitrou in bpo-20421.)

The `SSLSocket` class now implements a `SSLSocket.sendfile()` method. (Contributed by Giampaolo Rodola’ in bpo-17552.)

The `SSLSocket.send()` method now raises either the `ssl.SSLWantReadError` or `ssl.SSLWantWriteError` exception on a non-blocking socket if the operation would block. Previously, it would return 0. (Contributed by Nikolaus Rath in bpo-20951.)

The `cert_time_to_seconds()` function now interprets the input time as UTC and not as local time, per RFC 5280. Additionally, the return value is always an `int`. (Contributed by Akira Li in bpo-19940.)

New `SSLObject.shared_ciphers()` and `SSLSocket.shared_ciphers()` methods return the list of ciphers sent by the client during the handshake. (Contributed by Benjamin Peterson in bpo-23186.)

The `SSLSocket.do_handshake()`, `SSLSocket.read()`, `SSLSocket.shutdown()`, and `SSLSocket.write()` methods of the `SSLSocket` class no longer reset the socket timeout every time bytes are received or sent. The socket timeout is now the maximum total duration of the method. (Contributed by Victor Stinner in bpo-23853.)

The `match_hostname()` function now supports matching of IP addresses. (Contributed by Antoine Pitrou in bpo-23239.)
sqlite3

The Row class now fully supports the sequence protocol, in particular reversed() iteration and slice indexing. (Contributed by Claudiu Popa in bpo-10203; by Lucas Sinclair, Jessica McKellar, and Serhiy Storchaka in bpo-13583.)

subprocess

The new run() function has been added. It runs the specified command and returns a CompletedProcess object, which describes a finished process. The new API is more consistent and is the recommended approach to invoking subprocesses in Python code that does not need to maintain compatibility with earlier Python versions. (Contributed by Thomas Kluyver in bpo-23342.)

Examples:

```python
>>> subprocess.run(['ls', '-l'])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run('exit 1', shell=True, check=True)
Traceback (most recent call last):
... subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(['ls', '-l', '/dev/null'], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0, stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null')
```

sys
A new `set_coroutine_wrapper()` function allows setting a global hook that will be called whenever a coroutine object is created by an async def function. A corresponding `get_coroutine_wrapper()` can be used to obtain a currently set wrapper. Both functions are provisional, and are intended for debugging purposes only. (Contributed by Yury Selivanov in bpo-24017.)

A new `is_finalizing()` function can be used to check if the Python interpreter is shutting down. (Contributed by Antoine Pitrou in bpo-22696.)

**sysconfig**

The name of the user scripts directory on Windows now includes the first two components of the Python version. (Contributed by Paul Moore in bpo-23437.)

**tarfile**

The mode argument of the `open()` function now accepts "x" to request exclusive creation. (Contributed by Berker Peksag in bpo-21717.)

The `TarFile.extractall()` and `TarFile.extract()` methods now take a keyword argument `numeric_owner`. If set to `True`, the extracted files and directories will be owned by the numeric uid and gid from the tarfile. If set to `False` (the default, and the behavior in versions prior to 3.5), they will be owned by the named user and group in the tarfile. (Contributed by Michael Vogt and Eric Smith in bpo-23193.)
The `TarFile.list()` now accepts an optional `members` keyword argument that can be set to a subset of the list returned by `TarFile.getmembers()`. (Contributed by Serhiy Storchaka in bpo-21549.)

**threading**

Both the `Lock.acquire()` and `RLock.acquire()` methods now use a monotonic clock for timeout management. (Contributed by Victor Stinner in bpo-22043.)

**time**

The `monotonic()` function is now always available. (Contributed by Victor Stinner in bpo-22043.)

**timeit**

A new command line option `-u` or `--unit=U` can be used to specify the time unit for the timer output. Supported options are `usec`, `msec`, or `sec`. (Contributed by Julian Gindi in bpo-18983.)

The `timeit()` function has a new `globals` parameter for specifying the namespace in which the code will be running. (Contributed by Ben Roberts in bpo-2527.)

**tkinter**

The `tkinter._fix` module used for setting up the Tcl/Tk environment on Windows has been replaced by a private function in
the _tkinter module which makes no permanent changes to environment variables. (Contributed by Zachary Ware in bpo-20035.)

traceback

New walk_stack() and walk_tb() functions to conveniently traverse frame and traceback objects. (Contributed by Robert Collins in bpo-17911.)

New lightweight classes: TracebackException, StackSummary, and FrameSummary. (Contributed by Robert Collins in bpo-17911.)

Both the print_tb() and print_stack() functions now support negative values for the limit argument. (Contributed by Dmitry Kazakov in bpo-22619.)

types

A new coroutine() function to transform generator and generator-like objects into awaitables. (Contributed by Yury Selivanov in bpo-24017.)

A new type called CoroutineType, which is used for coroutine objects created by async def functions. (Contributed by Yury Selivanov in bpo-24400.)

unicodedata

The unicodedata module now uses data from Unicode 8.0.0.

unittest
The `TestLoader.loadTestsFromModule()` method now accepts a keyword-only argument `pattern` which is passed to `load_tests` as the third argument. Found packages are now checked for `load_tests` regardless of whether their path matches `pattern`, because it is impossible for a package name to match the default pattern. (Contributed by Robert Collins and Barry A. Warsaw in bpo-16662.)

Unittest discovery errors now are exposed in the `TestLoader.errors` attribute of the `TestLoader` instance. (Contributed by Robert Collins in bpo-19746.)

A new command line option `--locals` to show local variables in tracebacks. (Contributed by Robert Collins in bpo-22936.)

**unittest.mock**

The `Mock` class has the following improvements:

- The class constructor has a new `unsafe` parameter, which causes mock objects to raise `AttributeError` on attribute names starting with "assert". (Contributed by Kushal Das in bpo-21238.)
- A new `Mock.assert_not_called()` method to check if the mock object was called. (Contributed by Kushal Das in bpo-21262.)

The `MagicMock` class now supports `__truediv__()`, `__divmod__()` and `__matmul__()` operators. (Contributed by Johannes Baiter in bpo-20968, and Håkan Lövdahl in bpo-23581 and bpo-23568.)

It is no longer necessary to explicitly pass `create=True` to the
patch() function when patching builtin names. (Contributed by Kushal Das in bpo-17660.)

urllib

A new request.HTTPPasswordMgrWithPriorAuth class allows HTTP Basic Authentication credentials to be managed so as to eliminate unnecessary 401 response handling, or to unconditionally send credentials on the first request in order to communicate with servers that return a 404 response instead of a 401 if the Authorization header is not sent. (Contributed by Matej Cepl in bpo-19494 and Akshit Khurana in bpo-7159.)

A new quote_via argument for the parse.urlencode() function provides a way to control the encoding of query parts if needed. (Contributed by Samwyse and Arnon Yaari in bpo-13866.)

The request.urlopen() function accepts an ssl.SSLContext object as a context argument, which will be used for the HTTPS connection. (Contributed by Alex Gaynor in bpo-22366.)

The parse.urljoin() was updated to use the RFC 3986 semantics for the resolution of relative URLs, rather than RFC 1808 and RFC 2396. (Contributed by Demian Brecht and Senthil Kumaran in bpo-22118.)

wsgiref

The headers argument of the headers.Headers class constructor is now optional. (Contributed by Pablo Torres Navarrete and SilentGhost in bpo-5800.)
xmlrpc

The `client.ServerProxy` class now supports the context manager protocol. (Contributed by Claudiu Popa in bpo-20627.)

The `client.ServerProxy` constructor now accepts an optional `ssl.SSLContext` instance. (Contributed by Alex Gaynor in bpo-22960.)

xml.sax

SAX parsers now support a character stream of the `xmlreader.InputSource` object. (Contributed by Serhiy Storchaka in bpo-2175.)

`parseString()` now accepts a `str` instance. (Contributed by Serhiy Storchaka in bpo-10590.)

zipfile

ZIP output can now be written to unseekable streams. (Contributed by Serhiy Storchaka in bpo-23252.)

The `mode` argument of `ZipFile.open()` method now accepts "x" to request exclusive creation. (Contributed by Serhiy Storchaka in bpo-21717.)
Other module-level changes

Many functions in the `mmap`, `ossaudiodev`, `socket`, `ssl`, and `codecs` modules now accept writable bytes-like objects. (Contributed by Serhiy Storchaka in bpo-23001.)
Optimizations

The `os.walk()` function has been sped up by 3 to 5 times on POSIX systems, and by 7 to 20 times on Windows. This was done using the new `os.scandir()` function, which exposes file information from the underlying `readdir` or `FindFirstFile/FindNextFile` system calls. (Contributed by Ben Hoyt with help from Victor Stinner in `bpo-23605`.)

Construction of `bytes(int)` (filled by zero bytes) is faster and uses less memory for large objects. `calloc()` is used instead of `malloc()` to allocate memory for these objects. (Contributed by Victor Stinner in `bpo-21233`.)

Some operations on `ipaddress IPv4Network` and `IPv6Network` have been massively sped up, such as `subnets()`, `supernet()`, `summarize_address_range()`, `collapse_addresses()`. The speed up can range from 3 to 15 times. (Contributed by Antoine Pitrou, Michel Albert, and Markus in `bpo-21486`, `bpo-21487`, `bpo-20826`, `bpo-23266`.)

Pickling of `ipaddress` objects was optimized to produce significantly smaller output. (Contributed by Serhiy Storchaka in `bpo-23133`.)

Many operations on `io.BytesIO` are now 50% to 100% faster. (Contributed by Serhiy Storchaka in `bpo-15381` and David Wilson in `bpo-22003`.)

The `marshal.dumps()` function is now faster: 65–85% with versions 3 and 4, 20–25% with versions 0 to 2 on typical data, and up to 5 times in best cases. (Contributed by Serhiy Storchaka in `bpo-20416`.)
and bpo-23344.)

The UTF-32 encoder is now 3 to 7 times faster. (Contributed by Serhiy Storchaka in bpo-15027.)

Regular expressions are now parsed up to 10% faster. (Contributed by Serhiy Storchaka in bpo-19380.)

The json.dumps() function was optimized to run with ensure_ascii=False as fast as with ensure_ascii=True. (Contributed by Naoki Inada in bpo-23206.)

The PyObject_IsInstance() and PyObject_IsSubclass() functions have been sped up in the common case that the second argument has type as its metaclass. (Contributed Georg Brandl by in bpo-22540.)

Method caching was slightly improved, yielding up to 5% performance improvement in some benchmarks. (Contributed by Antoine Pitrou in bpo-22847.)

Objects from the random module now use 50% less memory on 64-bit builds. (Contributed by Serhiy Storchaka in bpo-23488.)

The property() getter calls are up to 25% faster. (Contributed by Joe Jevnik in bpo-23910.)

Instantiation of fractions.Fraction is now up to 30% faster. (Contributed by Stefan Behnel in bpo-22464.)

String methods find(), rfind(), split(), partition() and the in string operator are now significantly faster for searching 1-character substrings. (Contributed by Serhiy Storchaka in bpo-23573.)
Build and C API Changes

New calloc functions were added:

- `PyMem_RawCalloc()`,
- `PyMem_Calloc()`,
- `PyObject_Calloc()`.

(Contributed by Victor Stinner in bpo-21233.)

New encoding/decoding helper functions:

- `Py_DecodeLocale()` (replaced `_Py_char2wchar()`),
- `Py_EncodeLocale()` (replaced `_Py_wchar2char()`).

(Contributed by Victor Stinner in bpo-18395.)

A new `PyCodec_NameReplaceErrors()` function to replace the unicode encode error with \N{...} escapes. (Contributed by Serhiy Storchaka in bpo-19676.)

A new `PyErr_FormatV()` function similar to `PyErr_Format()`, but accepts a `va_list` argument. (Contributed by Antoine Pitrou in bpo-18711.)

A new `PyExc_RecursionError` exception. (Contributed by Georg Brandl in bpo-19235.)

New `PyModule_FromDefAndSpec()`, `PyModule_FromDefAndSpec2()`, and `PyModule_ExecDef()` functions introduced by PEP 489 – multi-phase extension module initialization. (Contributed by Petr Viktorin in bpo-24268.)
New \texttt{PyNumber\_MatrixMultiply()} and \texttt{PyNumber\_InPlaceMatrixMultiply()} functions to perform matrix multiplication. (Contributed by Benjamin Peterson in \texttt{bpo-21176}. See also \texttt{PEP 465} for details.)

The \texttt{PyTypeObject.tp\_finalize} slot is now part of the stable ABI.

Windows builds now require Microsoft Visual C++ 14.0, which is available as part of \texttt{Visual Studio 2015}.

Extension modules now include a platform information tag in their filename on some platforms (the tag is optional, and CPython will import extensions without it, although if the tag is present and mismatched, the extension won’t be loaded):

- On Linux, extension module filenames end with \texttt{.cpython-<major><minor>m-<architecture>-<os>.pyd}:
  - \texttt{<major>} is the major number of the Python version; for Python 3.5 this is 3.
  - \texttt{<minor>} is the minor number of the Python version; for Python 3.5 this is 5.
  - \texttt{<architecture>} is the hardware architecture the extension module was built to run on. It’s most commonly either \texttt{i386} for 32-bit Intel platforms or \texttt{x86\_64} for 64-bit Intel (and AMD) platforms.
  - \texttt{<os>} is always \texttt{linux-gnu}, except for extensions built to talk to the 32-bit ABI on 64-bit platforms, in which case it is \texttt{linux-gnu32} (and \texttt{<architecture>} will be \texttt{x86\_64}).

- On Windows, extension module filenames end with \texttt{<debug>.cp<major><minor>-<platform>.pyd}:
  - \texttt{<major>} is the major number of the Python version; for Python 3.5 this is 3.
- `<minor>` is the minor number of the Python version; for Python 3.5 this is 5.
- `<platform>` is the platform the extension module was built for, either `win32` for Win32, `win_amd64` for Win64, `win_ia64` for Windows Itanium 64, and `win_arm` for Windows on ARM.
- If built in debug mode, `<debug>` will be `_d`, otherwise it will be blank.
  - On OS X platforms, extension module filenames now end with `-darwin.so`.
  - On all other platforms, extension module filenames are the same as they were with Python 3.4.
Deprecated

New Keywords

async and await are not recommended to be used as variable, class, function or module names. Introduced by PEP 492 in Python 3.5, they will become proper keywords in Python 3.7.

Deprecated Python Behavior

Raising the StopIteration exception inside a generator will now generate a silent PendingDeprecationWarning, which will become a non-silent deprecation warning in Python 3.6 and will trigger a RuntimeError in Python 3.7. See PEP 479: Change StopIteration handling inside generators for details.

Unsupported Operating Systems

Windows XP is no longer supported by Microsoft, thus, per PEP 11, CPython 3.5 is no longer officially supported on this OS.

Deprecated Python modules, functions and methods

The formatter module has now graduated to full deprecation and is still slated for removal in Python 3.6.

The asyncio.async() function is deprecated in favor of ensure_future().
The *smtpd* module has in the past always decoded the DATA portion of email messages using the *utf-8* codec. This can now be controlled by the new *decode_data* keyword to *SMTPServer*. The default value is *True*, but this default is deprecated. Specify the *decode_data* keyword with an appropriate value to avoid the deprecation warning.

Directly assigning values to the *key*, *value* and *coded_value* of *http.cookies.Morsel* objects is deprecated. Use the *set()* method instead. In addition, the undocumented *LegalChars* parameter of *set()* is deprecated, and is now ignored.

Passing a format string as keyword argument *format_string* to the *format()* method of the *string.Formatter* class has been deprecated. (Contributed by Serhiy Storchaka in *bpo-23671*.)

The *platform.dist()* and *platform.linux_distribution()* functions are now deprecated. Linux distributions use too many different ways of describing themselves, so the functionality is left to a package. (Contributed by Vajrasky Kok and Berker Peksag in *bpo-1322*.)

The previously undocumented *from_function* and *from_builtin* methods of *inspect.Signature* are deprecated. Use the new *Signature.from_callable()* method instead. (Contributed by Yury Selivanov in *bpo-24248*.)

The *inspect.getargspec()* function is deprecated and scheduled to be removed in Python 3.6. (See *bpo-20438* for details.)

The *inspect.getfullargspec()* , *getcallargs()* , and *formatargspec()* functions are deprecated in favor of the
inspect.signature() API. (Contributed by Yury Selivanov in bpo-20438.)

getargvalues() and formatargvalues() functions were inadvertently marked as deprecated with the release of Python 3.5.0.

Use of re.LOCALE flag with str patterns or re.ASCII is now deprecated. (Contributed by Serhiy Storchaka in bpo-22407.)

Use of unrecognized special sequences consisting of '\' and an ASCII letter in regular expression patterns and replacement patterns now raises a deprecation warning and will be forbidden in Python 3.6. (Contributed by Serhiy Storchaka in bpo-23622.)

The undocumented and unofficial use_load_tests default argument of the unittest.TestLoader.loadTestsFromModule() method now is deprecated and ignored. (Contributed by Robert Collins and Barry A. Warsaw in bpo-16662.)
Removed

API and Feature Removals

The following obsolete and previously deprecated APIs and features have been removed:

- The `__version__` attribute has been dropped from the email package. The email code hasn’t been shipped separately from the stdlib for a long time, and the `__version__` string was not updated in the last few releases.
- The internal `Netrc` class in the `ftplib` module was deprecated in 3.4, and has now been removed. (Contributed by Matt Chaput in bpo-6623.)
- The concept of `.pyo` files has been removed.
- The `JoinableQueue` class in the provisional `asyncio` module was deprecated in 3.4.4 and is now removed. (Contributed by A. Jesse Jiryu Davis in bpo-23464.)
Porting to Python 3.5

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in Python behavior

- Due to an oversight, earlier Python versions erroneously accepted the following syntax:

```python
f(1 for x in [1], *args)
f(1 for x in [1], **kwargs)
```

Python 3.5 now correctly raises a `SyntaxError`, as generator expressions must be put in parentheses if not a sole argument to a function.

Changes in the Python API

- **PEP 475**: System calls are now retried when interrupted by a signal instead of raising `InterruptedError` if the Python signal handler does not raise an exception.
- Before Python 3.5, a `datetime.time` object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See `bpo-13936` for full details.
- The `ssl.SSLSocket.send()` method now raises either `ssl.SSLWantReadError` or `ssl.SSLWantWriteError` on a non-blocking socket if the operation would block. Previously, it would return 0. (Contributed by Nikolaus Rath in `bpo-20951`.)
- The `__name__` attribute of generators is now set from the
function name, instead of being set from the code name. Use `gen.gi_code.co_name` to retrieve the code name. Generators also have a new `__qualname__` attribute, the qualified name, which is now used for the representation of a generator (`repr(gen)`). (Contributed by Victor Stinner in bpo-21205.)

- The deprecated “strict” mode and argument of `HTMLParser`, `HTMLParser.error()`, and the `HTMLParserError` exception have been removed. (Contributed by Ezio Melotti in bpo-15114.) The `convert_charrefs` argument of `HTMLParser` is now `True` by default. (Contributed by Berker Peksag in bpo-21047.)

- Although it is not formally part of the API, it is worth noting for porting purposes (ie: fixing tests) that error messages that were previously of the form “‘sometype’ does not support the buffer protocol” are now of the form “a bytes-like object is required, not ‘sometype’”. (Contributed by Ezio Melotti in bpo-16518.)

- If the current directory is set to a directory that no longer exists then `FileNotFoundError` will no longer be raised and instead `find_spec()` will return `None` without caching `None` in `sys.path_importer_cache`, which is different than the typical case (bpo-22834).

- HTTP status code and messages from `http.client` and `http.server` were refactored into a common `HTTPStatus` enum. The values in `http.client` and `http.server` remain available for backwards compatibility. (Contributed by Demian Brecht in bpo-21793.)

- When an import loader defines `importlib.machinery.Loader.exec_module()` it is now expected to also define `create_module()` (raises a `DeprecationWarning` now, will be an error in Python 3.6). If the loader inherits from `importlib.abc.Loader` then there is nothing to do, else simply define `create_module()` to return `None`. (Contributed by Brett Cannon in bpo-23014.)

- The `re.split()` function always ignored empty pattern
matches, so the "x*" pattern worked the same as "x+", and the "\b" pattern never worked. Now `re.split()` raises a warning if the pattern could match an empty string. For compatibility, use patterns that never match an empty string (e.g. "x+" instead of "x*"). Patterns that could only match an empty string (such as "\b") now raise an error. (Contributed by Serhiy Storchaka in bpo-22818.)

- The `http.cookies.Morsel` dict-like interface has been made self-consistent: morsel comparison now takes the `key` and `value` into account, `copy()` now results in a `Morsel` instance rather than a `dict`, and `update()` will now raise an exception if any of the keys in the update dictionary are invalid. In addition, the undocumented `LegalChars` parameter of `set()` is deprecated and is now ignored. (Contributed by Demian Brecht in bpo-2211.)

- **PEP 488** has removed `.pyo` files from Python and introduced the optional `opt-` tag in `.pyc` file names. The `importlib.util.cache_from_source()` has gained an `optimization` parameter to help control the `opt-` tag. Because of this, the `debug_override` parameter of the function is now deprecated. `.pyo` files are also no longer supported as a file argument to the Python interpreter and thus serve no purpose when distributed on their own (i.e. sourceless code distribution). Due to the fact that the magic number for bytecode has changed in Python 3.5, all old `.pyo` files from previous versions of Python are invalid regardless of this PEP.

- The `socket` module now exports the `CAN_RAW_FD_FRAMES` constant on linux 3.6 and greater.

- The `ssl.cert_time_to_seconds()` function now interprets the input time as UTC and not as local time, per RFC 5280. Additionally, the return value is always an `int`. (Contributed by Akira Li in bpo-19940.)

- The `pygettext.py` Tool now uses the standard +NNNNN format for timezones in the POT-Creation-Date header.
• The `smtplib` module now uses `sys.stderr` instead of the previous module-level `stderr` variable for debug output. If your (test) program depends on patching the module-level variable to capture the debug output, you will need to update it to capture `sys.stderr` instead.

• The `str.startswith()` and `str.endswith()` methods no longer return `True` when finding the empty string and the indexes are completely out of range. (Contributed by Serhiy Storchaka in bpo-24284.)

• The `inspect.getdoc()` function now returns documentation strings inherited from base classes. Documentation strings no longer need to be duplicated if the inherited documentation is appropriate. To suppress an inherited string, an empty string must be specified (or the documentation may be filled in). This change affects the output of the `pydoc` module and the `help()` function. (Contributed by Serhiy Storchaka in bpo-15582.)

• Nested `functools.partial()` calls are now flattened. If you were relying on the previous behavior, you can now either add an attribute to a `functools.partial()` object or you can create a subclass of `functools.partial()`. (Contributed by Alexander Belopolsky in bpo-7830.)

Changes in the C API

• The undocumented format member of the (non-public) `PyMemoryViewObject` structure has been removed. All extensions relying on the relevant parts in `memoryobject.h` must be rebuilt.

• The `PyMemAllocator` structure was renamed to `PyMemAllocatorEx` and a new `calloc` field was added.

• Removed non-documentation macro `PyObject_REPR` which leaked references. Use format character `%R` in
PyUnicode_FromFormat() -like functions to format the `repr()` of the object. (Contributed by Serhiy Storchaka in bpo-22453.)

- Because the lack of the `__module__` attribute breaks pickling and introspection, a deprecation warning is now raised for builtin types without the `__module__` attribute. This would be an AttributeError in the future. (Contributed by Serhiy Storchaka in bpo-20204.)

- As part of the PEP 492 implementation, the `tp_reserved` slot of `PyTypeObject` was replaced with a `tp_as_async` slot. Refer to Coroutine Objects for new types, structures and functions.
Notable changes in Python 3.5.4

New make regen-all build target

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new make regen-all command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see Makefile.pre.in for details.

(Contributed by Victor Stinner in bpo-23404.)

New in version 3.5.4.

Removal of make touch build target

The make touch build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new make regen-all target.

(Contributed by Victor Stinner in bpo-23404.)

Changed in version 3.5.4.
What’s New In Python 3.4

Author: R. David Murray <rdmurray@bitdance.com> (Editor)

This article explains the new features in Python 3.4, compared to 3.3. Python 3.4 was released on March 16, 2014. For full details, see the changelog.

See also: PEP 429 – Python 3.4 Release Schedule
Summary – Release Highlights

New syntax features:

- No new syntax features were added in Python 3.4.

Other new features:

- pip should always be available (PEP 453).
- Newly created file descriptors are non-inheritable (PEP 446).
- command line option for isolated mode (bpo-16499).
- improvements in the handling of codecs that are not text encodings (multiple issues).
- A ModuleSpec Type for the Import System (PEP 451). (Affects importer authors.)
- The marshal format has been made more compact and efficient (bpo-16475).

New library modules:

- asyncio: New provisional API for asynchronous IO (PEP 3156).
- ensurepip: Bootstrapping the pip installer (PEP 453).
- enum: Support for enumeration types (PEP 435).
- pathlib: Object-oriented filesystem paths (PEP 428).
- selectors: High-level and efficient I/O multiplexing, built upon the select module primitives (part of PEP 3156).
- statistics: A basic numerically stable statistics library (PEP 450).
- tracemalloc: Trace Python memory allocations (PEP 454).

Significantly improved library modules:

- Single-dispatch generic functions in functools (PEP 443).
- New pickle protocol 4 (PEP 3154).
- `multiprocessing` now has an option to avoid using `os.fork` on Unix (bpo-8713).
- `email` has a new submodule, `contentmanager`, and a new `Message` subclass (`EmailMessage`) that simplify MIME handling (bpo-18891).
- The `inspect` and `pydoc` modules are now capable of correct introspection of a much wider variety of callable objects, which improves the output of the Python `help()` system.
- The `ipaddress` module API has been declared stable.

**Security improvements:**

- Secure and interchangeable hash algorithm (PEP 456).
- Make newly created file descriptors non-inheritable (PEP 446) to avoid leaking file descriptors to child processes.
- New command line option for `isolated mode`, (bpo-16499).
- `multiprocessing` now has an option to avoid using `os.fork` on Unix. `spawn` and `forkserver` are more secure because they avoid sharing data with child processes.
- `multiprocessing` child processes on Windows no longer inherit all of the parent's inheritable handles, only the necessary ones.
- A new `hashlib.pbkdf2_hmac()` function provides the PKCS#5 password-based key derivation function 2.
- TLSv1.1 and TLSv1.2 support for `ssl`.
- Retrieving certificates from the Windows system cert store support for `ssl`.
- Server-side SNI (Server Name Indication) support for `ssl`.
- The `ssl.SSLContext` class has a lot of improvements.
- All modules in the standard library that support SSL now support server certificate verification, including hostname matching (`ssl.match_hostname()`) and CRLs (Certificate Revocation lists, see `ssl.SSLContext.load_verify_locations()`).
CPython implementation improvements:

- **Safe object finalization** ([PEP 442](https://www.python.org/dev/peps/pep-0442/)).
- Leveraging **PEP 442**, in most cases module globals are no longer set to None during finalization ([bpo-18214](https://bugs.python.org/issue18214)).
- **Configurable memory allocators** ([PEP 445](https://www.python.org/dev/peps/pep-0445/)).
- **Argument Clinic** ([PEP 436](https://www.python.org/dev/peps/pep-0436/)).

Please read on for a comprehensive list of user-facing changes, including many other smaller improvements, CPython optimizations, deprecations, and potential porting issues.
New Features

PEP 453: Explicit Bootstrapping of PIP in Python Installations

Bootstrapping pip By Default

The new `ensurepip` module (defined in PEP 453) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations and virtual environments. The version of `pip` included with Python 3.4.0 is `pip 1.5.4`, and future 3.4.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies. On Windows and in virtual environments on all platforms, the unversioned `pip` command will also be installed. On other platforms, the system wide unversioned `pip` command typically refers to the separately installed Python 2 version.

The `pyvenv` command line utility and the `venv` module make use of the `ensurepip` module to make `pip` readily available in virtual environments. When using the command line utility, `pip` is installed by default, while when using the `venv` module API installation of `pip` must be requested explicitly.

For CPython source builds on POSIX systems, the `make install` and `make altinstall` commands bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden
through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As discussed in the PEP, platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

**Note:** To avoid conflicts between parallel Python 2 and Python 3 installations, only the versioned `pip3` and `pip3.4` commands are bootstrapped by default when `ensurepip` is invoked directly - the `-default-pip` option is needed to also request the unversioned `pip` command. `pyvenv` and the Windows installer ensure that the unqualified `pip` command is made available in those environments, and `pip` can always be invoked via the `-m` switch rather than directly to avoid ambiguity on systems with multiple Python installations.

**Documentation Changes**

As part of this change, the Installing Python Modules and Distributing Python Modules sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide and the documentation of the individual projects.
However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as Installing Python Modules (Legacy version) and Distributing Python Modules (Legacy version).

See also:

PEP 453 – Explicit bootstrapping of pip in Python installations
PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

PEP 446: Newly Created File Descriptors Are Non-Inheritable

PEP 446 makes newly created file descriptors non-inheritable. In general, this is the behavior an application will want: when launching a new process, having currently open files also open in the new process can lead to all sorts of hard to find bugs, and potentially to security issues.

However, there are occasions when inheritance is desired. To support these cases, the following new functions and methods are available:

- `os.get_inheritable()`, `os.set_inheritable()`
- `os.get_handle_inheritable()`, `os.set_handle_inheritable()`
- `socket.socket.get_inheritable()`, `socket.socket.set_inheritable()`

See also:

PEP 446 – Make newly created file descriptors non-inheritable
Improvements to Codec Handling

Since it was first introduced, the `codecs` module has always been intended to operate as a type-neutral dynamic encoding and decoding system. However, its close coupling with the Python text model, especially the type restricted convenience methods on the built-in `str`, `bytes` and `bytearray` types, has historically obscured that fact.

As a key step in clarifying the situation, the `codecs.encode()` and `codecs.decode()` convenience functions are now properly documented in Python 2.7, 3.3 and 3.4. These functions have existed in the `codecs` module (and have been covered by the regression test suite) since Python 2.4, but were previously only discoverable through runtime introspection.

Unlike the convenience methods on `str`, `bytes` and `bytearray`, the `codecs` convenience functions support arbitrary codecs in both Python 2 and Python 3, rather than being limited to Unicode text encodings (in Python 3) or `basestring` <-> `basestring` conversions (in Python 2).

In Python 3.4, the interpreter is able to identify the known non-text encodings provided in the standard library and direct users towards these general purpose convenience functions when appropriate:

```python
>>> b"abcdef".decode("hex")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs
```
In a related change, whenever it is feasible without breaking backwards compatibility, exceptions raised during encoding and decoding operations are wrapped in a chained exception of the same type that mentions the name of the codec responsible for producing the error:

```python
>>> import codecs

>>> codecs.decode(b"abcdefgh", "hex")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/hex_codec.py", line 128, in <module>
    return (binascii.a2b_hex(input), len(input))
binascii.Error: Non-hexadecimal digit found
```

The above exception was the direct cause of the following exception:

```python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
binascii.Error: decoding with 'hex' codec failed (Error: Non-hexadecimal digit found)
```

```python
>>> codecs.encode("hello", "bz2")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/bz2_codec.py", line 128, in <module>
    return (bz2.compress(input), len(input))
File "/usr/lib/python3.4/bz2.py", line 498, in compress
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: encoding with 'bz2' codec failed (TypeError: 'str' does not support the buffer interface)
```

Finally, as the examples above show, these improvements have permitted the restoration of the convenience aliases for the non-Unicode codecs that were themselves restored in Python 3.2. This means that encoding binary data to and from its hexadecimal representation (for example) can now be written as:

```
>>> from codecs import encode, decode
>>> encode(b"hello", "hex")
b'68656c6c6f'
>>> decode(b'68656c6c6f", "hex")
b'hello'
```

The binary and text transforms provided in the standard library are detailed in Binary Transforms and Text Transforms.

(Contributed by Nick Coghlan in bpo-7475, bpo-17827, bpo-17828 and bpo-19619.)

**PEP 451: A ModuleSpec Type for the Import System**

**PEP 451** provides an encapsulation of the information about a module that the import machinery will use to load it (that is, a module specification). This helps simplify both the import implementation and
several import-related APIs. The change is also a stepping stone for several future import-related improvements.

The public-facing changes from the PEP are entirely backward-compatible. Furthermore, they should be transparent to everyone but importer authors. Key finder and loader methods have been deprecated, but they will continue working. New importers should use the new methods described in the PEP. Existing importers should be updated to implement the new methods. See the Deprecated section for a list of methods that should be replaced and their replacements.

Other Language Changes

Some smaller changes made to the core Python language are:

- Unicode database updated to UCD version 6.3.
- `min()` and `max()` now accept a default keyword-only argument that can be used to specify the value they return if the iterable they are evaluating has no elements. (Contributed by Julian Berman in bpo-18111.)
- Module objects are now weakref'able.
- Module `__file__` attributes (and related values) should now always contain absolute paths by default, with the sole exception of `__main__.__file__` when a script has been executed directly using a relative path. (Contributed by Brett Cannon in bpo-18416.)
- All the UTF-* codecs (except UTF-7) now reject surrogates during both encoding and decoding unless the surrogatepass error handler is used, with the exception of the UTF-16 decoder (which accepts valid surrogate pairs) and the UTF-16 encoder (which produces them while encoding non-BMP characters). (Contributed by Victor Stinner, Kang-Hao (Kenny) Lu and Serhiy Storchaka in bpo-12892.)
• New German EBCDIC codec cp273. (Contributed by Michael Bierenfeld and Andrew Kuchling in bpo-1097797.)
• New Ukrainian codec cp1125. (Contributed by Serhiy Storchaka in bpo-19668.)
• bytes.join() and bytearray.join() now accept arbitrary buffer objects as arguments. (Contributed by Antoine Pitrou in bpo-15958.)
• The int constructor now accepts any object that has an __index__ method for its base argument. (Contributed by Mark Dickinson in bpo-16772.)
• Frame objects now have a clear() method that clears all references to local variables from the frame. (Contributed by Antoine Pitrou in bpo-17934.)
• memoryview is now registered as a Sequence, and supports the reversed() builtin. (Contributed by Nick Coghlan and Claudiu Popa in bpo-18690 and bpo-19078.)
• Signatures reported by help() have been modified and improved in several cases as a result of the introduction of Argument Clinic and other changes to the inspect and pydoc modules.
• __length_hint__() is now part of the formal language specification (see PEP 424). (Contributed by Armin Ronacher in bpo-16148.)
New Modules

asyncio

The new asyncio module (defined in PEP 3156) provides a standard pluggable event loop model for Python, providing solid asynchronous IO support in the standard library, and making it easier for other event loop implementations to interoperate with the standard library and each other.

For Python 3.4, this module is considered a provisional API.

See also:

**PEP 3156** – Asynchronous IO Support Rebooted: the “asyncio” Module

PEP written and implementation led by Guido van Rossum.

ensurepip

The new ensurepip module is the primary infrastructure for the PEP 453 implementation. In the normal course of events end users will not need to interact with this module, but it can be used to manually bootstrap pip if the automated bootstrapping into an installation or virtual environment was declined.

ensurepip includes a bundled copy of pip, up-to-date as of the first release candidate of the release of CPython with which it ships (this applies to both maintenance releases and feature releases). ensurepip does not access the internet. If the installation has Internet access, after ensurepip is run the bundled pip can be used
to upgrade `pip` to a more recent release than the bundled one. (Note that such an upgraded version of `pip` is considered to be a separately installed package and will not be removed if Python is uninstalled.)

The module is named `ensurepip` because if called when `pip` is already installed, it does nothing. It also has an `--upgrade` option that will cause it to install the bundled copy of `pip` if the existing installed version of `pip` is older than the bundled copy.

**enum**

The new `enum` module (defined in [PEP 435](https://www.python.org/dev/peps/pep-0435/)) provides a standard implementation of enumeration types, allowing other modules (such as `socket`) to provide more informative error messages and better debugging support by replacing opaque integer constants with backwards compatible enumeration values.

**See also:**

[PEP 435 – Adding an Enum type to the Python standard library](https://www.python.org/dev/peps/pep-0435/)

PEP written by Barry Warsaw, Eli Bendersky and Ethan Furman, implemented by Ethan Furman.

**pathlib**

The new `pathlib` module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between `pure paths`, which provide purely computational operations without I/O, and `concrete paths`, which inherit from pure paths but also provide I/O operations.
For Python 3.4, this module is considered a provisional API.

See also:

**PEP 428** – The pathlib module – object-oriented filesystem paths  
PEP written and implemented by Antoine Pitrou.

selectors

The new **selectors** module (created as part of implementing **PEP 3156**) allows high-level and efficient I/O multiplexing, built upon the **select** module primitives.

statistics

The new **statistics** module (defined in **PEP 450**) offers some core statistics functionality directly in the standard library. This module supports calculation of the mean, median, mode, variance and standard deviation of a data series.

See also:

**PEP 450** – Adding A Statistics Module To The Standard Library  
PEP written and implemented by Steven D’Aprano

tracemalloc

The new **tracemalloc** module (defined in **PEP 454**) is a debug tool to trace memory blocks allocated by Python. It provides the following information:
• Trace where an object was allocated
• Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
• Compute the differences between two snapshots to detect memory leaks

See also:

**PEP 454** – Add a new tracemalloc module to trace Python memory allocations
  PEP written and implemented by Victor Stinner
Improved Modules

abc

New function `abc.get_cache_token()` can be used to know when to invalidate caches that are affected by changes in the object graph. (Contributed by Łukasz Langa in bpo-16832.)

New class `ABC` has `ABCMeta` as its meta class. Using `ABC` as a base class has essentially the same effect as specifying `metaclass=abc.ABCMeta`, but is simpler to type and easier to read. (Contributed by Bruno Dupuis in bpo-16049.)

aifc

The `getparams()` method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-17818.)

`aifc.open()` now supports the context management protocol: when used in a `with` block, the `close()` method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchacha in bpo-16486.)

The `writeframesraw()` and `writeframes()` methods now accept any `bytes-like object`. (Contributed by Serhiy Storchaka in bpo-8311.)

argparse

The `FileType` class now accepts `encoding` and `errors` arguments, which are passed through to `open()`. (Contributed by Lucas Maystre in bpo-11175.)
**audioop**

*audioop* now supports 24-bit samples. (Contributed by Serhiy Storchaka in *bpo-12866*.)

New *byteswap()* function converts big-endian samples to little-endian and vice versa. (Contributed by Serhiy Storchaka in *bpo-19641*.)

All *audioop* functions now accept any *bytes-like object*. Strings are not accepted: they didn’t work before, now they raise an error right away. (Contributed by Serhiy Storchaka in *bpo-16685*.)

**base64**

The encoding and decoding functions in *base64* now accept any *bytes-like object* in cases where it previously required a *bytes* or *bytearray* instance. (Contributed by Nick Coghlan in *bpo-17839*.)

New functions *a85encode()*, *a85decode()*, *b85encode()*, and *b85decode()* provide the ability to encode and decode binary data from and to *Ascii85* and the *git/mercurial Base85* formats, respectively. The *a85* functions have options that can be used to make them compatible with the variants of the *Ascii85* encoding, including the Adobe variant. (Contributed by Martin Morrison, the Mercurial project, Serhiy Storchaka, and Antoine Pitrou in *bpo-17618*.)

**collections**

The *ChainMap.new_child()* method now accepts an *m* argument specifying the child map to add to the chain. This allows an existing
mapping and/or a custom mapping type to be used for the child. (Contributed by Vinay Sajip in bpo-16613.)

colorsys

The number of digits in the coefficients for the RGB — YIQ conversions have been expanded so that they match the FCC NTSC versions. The change in results should be less than 1% and may better match results found elsewhere. (Contributed by Brian Landers and Serhiy Storchaka in bpo-14323.)

contextlib

The new contextlib.suppress context manager helps to clarify the intent of code that deliberately suppresses exceptions from a single statement. (Contributed by Raymond Hettinger in bpo-15806 and Zero Piraeus in bpo-19266.)

The new contextlib.redirect_stdout() context manager makes it easier for utility scripts to handle inflexible APIs that write their output to sys.stdout and don’t provide any options to redirect it. Using the context manager, the sys.stdout output can be redirected to any other stream or, in conjunction with io.StringIO, to a string. The latter can be especially useful, for example, to capture output from a function that was written to implement a command line interface. It is recommended only for utility scripts because it affects the global state of sys.stdout. (Contributed by Raymond Hettinger in bpo-15805.)

The contextlib documentation has also been updated to include a discussion of the differences between single use, reusable and
reentrant context managers.

**dbm**

dbm.open() objects now support the context management protocol. When used in a `with` statement, the `close` method of the database object will be called automatically at the end of the block. (Contributed by Claudiu Popa and Nick Coghlan in [bpo-19282](https://bugs.python.org/issue19282).)

**dis**

Functions `show_code()`, `dis()`, `distb()`, and `disassemble()` now accept a keyword-only `file` argument that controls where they write their output.

The `dis` module is now built around an `Instruction` class that provides object oriented access to the details of each individual bytecode operation.

A new method, `get_instructions()`, provides an iterator that emits the Instruction stream for a given piece of Python code. Thus it is now possible to write a program that inspects and manipulates a bytecode object in ways different from those provided by the `dis` module itself. For example:

```python
>>> import dis
>>> for instr in dis.get_instructions(lambda x: x + ... print(instr.opname)
LOAD_FAST
LOAD_CONST
BINARY_ADD
RETURN_VALUE
```
The various display tools in the `dis` module have been rewritten to use these new components.

In addition, a new application-friendly class `Bytecode` provides an object-oriented API for inspecting bytecode in both in human-readable form and for iterating over instructions. The `Bytecode` constructor takes the same arguments that `get_instruction()` does (plus an optional `current_offset`), and the resulting object can be iterated to produce `Instruction` objects. But it also has a `dis` method, equivalent to calling `dis` on the constructor argument, but returned as a multi-line string:

```
>>> bytecode = dis.Bytecode(lambda x: x + 1, current_offset=123)
```  
```
>>> for instr in bytecode:
...    print('{0} ({1})'.format(instr.opname, instr.opcode))
```

```
LOAD_FAST (124)
LOAD_CONST (100)
BINARY_ADD (23)
RETURN_VALUE (83)
```  

```
>>> bytecode.dis().splitlines()
[ ' 1   0 LOAD_FAST 0 (x)',
  ' --> 3 LOAD_CONST 1 (1)',
  ' 6 BINARY_ADD',
  ' 7 RETURN_VALUE']
```  

`Bytecode` also has a class method, `from_traceback()`, that provides the ability to manipulate a traceback (that is, `print(Bytecode.from_traceback(tb).dis())` is equivalent to `distsb(tb)`).

(Contributed by Nick Coghlan, Ryan Kelly and Thomas Kluyver in `bpo-11816` and Claudiu Popa in `bpo-17916`.)
New function `stack_effect()` computes the effect on the Python stack of a given opcode and argument, information that is not otherwise available. (Contributed by Larry Hastings in bpo-19722.)

**doctest**

A new option flag, `FAIL_FAST`, halts test running as soon as the first failure is detected. (Contributed by R. David Murray and Daniel Urban in bpo-16522.)

The **doctest** command line interface now uses **argparse**, and has two new options, `-o` and `-f`. `-o` allows **doctest options** to be specified on the command line, and `-f` is a shorthand for `-o FAIL_FAST` (to parallel the similar option supported by the **unittest** CLI). (Contributed by R. David Murray in bpo-11390.)

**doctest** will now find doctests in extension module `__doc__` strings. (Contributed by Zachary Ware in bpo-3158.)

**email**

**as_string()** now accepts a `policy` argument to override the default policy of the message when generating a string representation of it. This means that **as_string** can now be used in more circumstances, instead of having to create and use a **generator** in order to pass formatting parameters to its `flatten` method. (Contributed by R. David Murray in bpo-18600.)

New method **as_bytes()** added to produce a bytes representation of the message in a fashion similar to how **as_string** produces a string representation. It does not accept the `maxheaderlen` argument,
but does accept the `unixfrom` and `policy` arguments. The `Message.__bytes__()` method calls it, meaning that `bytes(mymsg)` will now produce the intuitive result: a bytes object containing the fully formatted message. (Contributed by R. David Murray in bpo-18600.)

The `Message.set_param()` message now accepts a `replace` keyword argument. When specified, the associated header will be updated without changing its location in the list of headers. For backward compatibility, the default is `False`. (Contributed by R. David Murray in bpo-18891.)

A pair of new subclasses of `Message` have been added (`EmailMessage` and `MIMEPart`), along with a new sub-module, `contentmanager` and a new `policy` attribute `content_manager`. All documentation is currently in the new module, which is being added as part of email's new [provisional API](https://docs.python.org/3/library/email.html). These classes provide a number of new methods that make extracting content from and inserting content into email messages much easier. For details, see the `contentmanager` documentation and the email: Examples. These API additions complete the bulk of the work that was planned as part of the email6 project. The currently provisional API is scheduled to become final in Python 3.5 (possibly with a few minor additions in the area of error handling). (Contributed by R. David Murray in bpo-18891.)

**filecmp**

A new `clear_cache()` function provides the ability to clear the `filecmp` comparison cache, which uses `os.stat()` information to determine if the file has changed since the last compare. This can be used, for example, if the file might have been changed and re-checked.
in less time than the resolution of a particular filesystem’s file modification time field. (Contributed by Mark Levitt in bpo-18149.)

New module attribute DEFAULT_IGNORES provides the list of directories that are used as the default value for the ignore parameter of the dircmp() function. (Contributed by Eli Bendersky in bpo-15442.)

**functools**

The new partialmethod() descriptor brings partial argument application to descriptors, just as partial() provides for normal callables. The new descriptor also makes it easier to get arbitrary callables (including partial() instances) to behave like normal instance methods when included in a class definition. (Contributed by Alon Horev and Nick Coghlan in bpo-4331.)

The new singledispatch() decorator brings support for single-dispatch generic functions to the Python standard library. Where object oriented programming focuses on grouping multiple operations on a common set of data into a class, a generic function focuses on grouping multiple implementations of an operation that allows it to work with different kinds of data.

**See also:**

**PEP 443** – Single-dispatch generic functions
PEP written and implemented by Łukasz Langa.

total_ordering() now supports a return value of NotImplemented from the underlying comparison function.
A pure-python version of the `partial()` function is now in the stdlib; in CPython it is overridden by the C accelerated version, but it is available for other implementations to use. (Contributed by Brian Thorne in bpo-12428.)

**gc**

New function `get_stats()` returns a list of three per-generation dictionaries containing the collections statistics since interpreter startup. (Contributed by Antoine Pitrou in bpo-16351.)

**glob**

A new function `escape()` provides a way to escape special characters in a filename so that they do not become part of the globbing expansion but are instead matched literally. (Contributed by Serhiy Storchaka in bpo-8402.)

**hashlib**

A new `hashlib.pbkdf2_hmac()` function provides the PKCS#5 password-based key derivation function 2. (Contributed by Christian Heimes in bpo-18582.)

The `name` attribute of `hashlib` hash objects is now a formally supported interface. It has always existed in CPython’s `hashlib` (although it did not return lower case names for all supported hashes), but it was not a public interface and so some other Python implementations have not previously supported it. (Contributed by
hmac

hmac now accepts `bytearray` as well as `bytes` for the `key` argument to the `new()` function, and the `msg` parameter to both the `new()` function and the `update()` method now accepts any type supported by the `hashlib` module. (Contributed by Jonas Borgström in bpo-18240.)

The `digestmod` argument to the `hmac.new()` function may now be any hash digest name recognized by `hashlib`. In addition, the current behavior in which the value of `digestmod` defaults to MD5 is deprecated: in a future version of Python there will be no default value. (Contributed by Christian Heimes in bpo-17276.)

With the addition of `block_size` and `name` attributes (and the formal documentation of the `digest_size` attribute), the `hmac` module now conforms fully to the PEP 247 API. (Contributed by Christian Heimes in bpo-18775.)

html

New function `unescape()` function converts HTML5 character references to the corresponding Unicode characters. (Contributed by Ezio Melotti in bpo-2927.)

`HTMLParser` accepts a new keyword argument `convert_charrefs` that, when `True`, automatically converts all character references. For backward-compatibility, its value defaults to `False`, but it will change to `True` in a future version of Python, so you are invited to set it
explicitly and update your code to use this new feature. (Contributed by Ezio Melotti in bpo-13633.)

The strict argument of HTMLParser is now deprecated. (Contributed by Ezio Melotti in bpo-15114.)

http

send_error() now accepts an optional additional explain parameter which can be used to provide an extended error description, overriding the hardcoded default if there is one. This extended error description will be formatted using the error_message_format attribute and sent as the body of the error response. (Contributed by Karl Cow in bpo-12921.)

The http.server command line interface now has a -b/--bind option that causes the server to listen on a specific address. (Contributed by Malte Swart in bpo-17764.)

idlelib and IDLE

Since idlelib implements the IDLE shell and editor and is not intended for import by other programs, it gets improvements with every release. See Lib/idlelib/NEWS.txt for a cumulative list of changes since 3.3.0, as well as changes made in future 3.4.x releases. This file is also available from the IDLE Help ▶ About IDLE dialog.

importlib

The InspectLoader ABC defines a new method, source_to_code() that accepts source data and a path and returns
a code object. The default implementation is equivalent to:

```
compile(data, path, 'exec', dont_inherit=True).
```

(Contributed by Eric Snow and Brett Cannon in bpo-15627.)

`InspectLoader` also now has a default implementation for the `get_code()` method. However, it will normally be desirable to override the default implementation for performance reasons. (Contributed by Brett Cannon in bpo-18072.)

The `reload()` function has been moved from `imp` to `importlib` as part of the `imp` module deprecation. (Contributed by Berker Peksag in bpo-18193.)

`importlib.util` now has a `MAGIC_NUMBER` attribute providing access to the bytecode version number. This replaces the `get_magic()` function in the deprecated `imp` module. (Contributed by Brett Cannon in bpo-18192.)

New `importlib.util` functions `cache_from_source()` and `source_from_cache()` replace the same-named functions in the deprecated `imp` module. (Contributed by Brett Cannon in bpo-18194.)

The `importlib` bootstrap NamespaceLoader now conforms to the `InspectLoader` ABC, which means that `runpy` and `python -m` can now be used with namespace packages. (Contributed by Brett Cannon in bpo-18058.)

`importlib.util` has a new function `decode_source()` that decodes source from bytes using universal newline processing. This is useful for implementing `InspectLoader.get_source()` methods.

`importlib.machinery.ExtensionFileLoader` now has a `get_filename()` method. This was inadvertently omitted in the
The `inspect` module now offers a basic command line interface to quickly display source code and other information for modules, classes and functions. (Contributed by Claudiu Popa and Nick Coghlan in bpo-18626.)

`unwrap()` makes it easy to unravel wrapper function chains created by `functools.wraps()` (and any other API that sets the `__wrapped__` attribute on a wrapper function). (Contributed by Daniel Urban, Aaron Iles and Nick Coghlan in bpo-13266.)

As part of the implementation of the new `enum` module, the `inspect` module now has substantially better support for custom `__dir__` methods and dynamic class attributes provided through metaclasses. (Contributed by Ethan Furman in bpo-18929 and bpo-19030.)

`getfullargspec()` and `getargspec()` now use the `signature()` API. This allows them to support a much broader range of callables, including those with `__signature__` attributes, those with metadata provided by argument clinic, `functools.partial()` objects and more. Note that, unlike `signature()`, these functions still ignore `__wrapped__` attributes, and report the already bound first argument for bound methods, so it is still necessary to update your code to use `signature()` directly if those features are desired. (Contributed by Yury Selivanov in bpo-17481.)

`signature()` now supports duck types of CPython functions, which adds support for functions compiled with Cython. (Contributed by
ipaddress

`ipaddress` was added to the standard library in Python 3.3 as a provisional API. With the release of Python 3.4, this qualification has been removed: `ipaddress` is now considered a stable API, covered by the normal standard library requirements to maintain backwards compatibility.

A new `is_global` property is `True` if an address is globally routeable. (Contributed by Peter Moody in bpo-17400.)

logging

The `TimedRotatingFileHandler` has a new `atTime` parameter that can be used to specify the time of day when rollover should happen. (Contributed by Ronald Oussoren in bpo-9556.)

`SocketHandler` and `DatagramHandler` now support Unix domain sockets (by setting `port` to `None`). (Contributed by Vinay Sajip in commit ce46195b56a9.)

`fileConfig()` now accepts a `configparser.RawConfigParser` subclass instance for the `fname` parameter. This facilitates using a configuration file when logging configuration is just a part of the overall application configuration, or where the application modifies the configuration before passing it to `fileConfig()`. (Contributed by Vinay Sajip in bpo-16110.)

Logging configuration data received from a socket via the `logging.config.listen()` function can now be validated before
being processed by supplying a verification function as the argument to the new `verify` keyword argument. (Contributed by Vinay Sajip in bpo-15452.)

**marshal**

The default `marshal` version has been bumped to 3. The code implementing the new version restores the Python2 behavior of recording only one copy of interned strings and preserving the interning on deserialization, and extends this “one copy” ability to any object type (including handling recursive references). This reduces both the size of `.pyc` files and the amount of memory a module occupies in memory when it is loaded from a `.pyc` (or `.pyo`) file. (Contributed by Kristján Valur Jónsson in bpo-16475, with additional speedups by Antoine Pitrou in bpo-19219.)

**mmap**

`mmap` objects can now be `weakref`ed. (Contributed by Valerie Lambert in bpo-4885.)

**multiprocessing**

On Unix two new `start methods`, `spawn` and `forkserver`, have been added for starting processes using `multiprocessing`. These make the mixing of processes with threads more robust, and the `spawn` method matches the semantics that multiprocessing has always used on Windows. New function `get_all_start_methods()` reports all start methods available on the platform, `get_start_method()` reports the current start method, and `set_start_method()` sets the
start method. (Contributed by Richard Oudkerk in bpo-8713.)

`multiprocessing` also now has the concept of a context, which
determines how child processes are created. New function
`get_context()` returns a context that uses a specified start method.
It has the same API as the `multiprocessing` module itself, so you
can use it to create `Pool` objects and other objects that will operate within
that context. This allows a framework and an application or different
parts of the same application to use multiprocessing without interfering
with each other. (Contributed by Richard Oudkerk in bpo-18999.)

Except when using the old `fork` start method, child processes no
longer inherit unneeded handles/file descriptors from their parents
(part of bpo-8713).

`multiprocessing` now relies on `runpy` (which implements the `-m
switch) to initialise `__main__` appropriately in child processes when
using the `spawn` or `forkserver` start methods. This resolves some
edge cases where combining multiprocessing, the `-m command line
switch, and explicit relative imports could cause obscure failures in
child processes. (Contributed by Nick Coghlan in bpo-19946.)

**operator**

New function `length_hint()` provides an implementation of the
specification for how the `__length_hint__()` special method
should be used, as part of the PEP 424 formal specification of this
language feature. (Contributed by Armin Ronacher in bpo-16148.)

There is now a pure-python version of the `operator` module
available for reference and for use by alternate implementations of
Python. (Contributed by Zachary Ware in bpo-16694.)
OS

There are new functions to get and set the inheritable flag of a file descriptor (`os.get_inheritable()`, `os.set_inheritable()`), or a Windows handle (`os.get_handle_inheritable()`, `os.set_handle_inheritable()`).

New function `cpu_count()` reports the number of CPUs available on the platform on which Python is running (or `None` if the count can’t be determined). The `multiprocessing.cpu_count()` function is now implemented in terms of this function). (Contributed by Trent Nelson, Yogesh Chaudhari, Victor Stinner, and Charles-François Natali in bpo-17914.)

`os.path.samestat()` is now available on the Windows platform (and the `os.path.samefile()` implementation is now shared between Unix and Windows). (Contributed by Brian Curtin in bpo-11939.)

`os.path.ismount()` now recognizes volumes mounted below a drive root on Windows. (Contributed by Tim Golden in bpo-9035.)

`os.open()` supports two new flags on platforms that provide them, `O_PATH` (un-opened file descriptor), and `O_TMPFILE` (unnamed temporary file; as of 3.4.0 release available only on Linux systems with a kernel version of 3.11 or newer that have uapi headers). (Contributed by Christian Heimes in bpo-18673 and Benjamin Peterson, respectively.)

`pdb`
**pdb** has been enhanced to handle generators, *yield*, and *yield from* in a more useful fashion. This is especially helpful when debugging *asyncio* based programs. (Contributed by Andrew Svetlov and Xavier de Gaye in [bpo-16596](https://bugs.python.org/issue16596).)

The *print* command has been removed from **pdb**, restoring access to the Python *print()* function from the **pdb** command line. **Python2**'s **pdb** did not have a *print* command; instead, entering *print* executed the *print* statement. In **Python3** *print* was mistakenly made an alias for the **pdb** *p* command. *p*, however, prints the *repr* of its argument, not the *str* like the **Python2** *print* command did. Worse, the **Python3** **pdb** *print* command shadowed the **Python3** *print* function, making it inaccessible at the **pdb** prompt. (Contributed by Connor Osborn in [bpo-18764](https://bugs.python.org/issue18764).)

**pickle**

**pickle** now supports (but does not use by default) a new pickle protocol, protocol 4. This new protocol addresses a number of issues that were present in previous protocols, such as the serialization of nested classes, very large strings and containers, and classes whose *__new__()* method takes keyword-only arguments. It also provides some efficiency improvements.

**See also:**

**PEP 3154** – Pickle protocol 4

PEP written by Antoine Pitrou and implemented by Alexandre Vassalotti.

**plistlib**
plistlib now has an API that is similar to the standard pattern for stdlib serialization protocols, with new `load()`, `dump()`, `loads()`, and `dumps()` functions. (The older API is now deprecated.) In addition to the already supported XML plist format (FMT_XML), it also now supports the binary plist format (FMT_BINARY). (Contributed by Ronald Oussoren and others in bpo-14455.)

poplib

Two new methods have been added to poplib: `capa()`, which returns the list of capabilities advertised by the POP server, and `stls()`, which switches a clear-text POP3 session into an encrypted POP3 session if the POP server supports it. (Contributed by Lorenzo Catucci in bpo-4473.)

pprint

The pprint module's PrettyPrinter class and its `pformat()`, and `pprint()` functions have a new option, `compact`, that controls how the output is formatted. Currently setting `compact` to True means that sequences will be printed with as many sequence elements as will fit within `width` on each (indented) line. (Contributed by Serhiy Storchaka in bpo-19132.) Long strings are now wrapped using Python's normal line continuation syntax. (Contributed by Antoine Pitrou in bpo-17150.)

pty

`pty.spawn()` now returns the status value from `os.waitpid()` on
the child process, instead of None. (Contributed by Gregory P. Smith.)

**pydoc**

The **pydoc** module is now based directly on the `inspect.signature()` introspection API, allowing it to provide signature information for a wider variety of callable objects. This change also means that `__wrapped__` attributes are now taken into account when displaying help information. (Contributed by Larry Hastings in bpo-19674.)

The **pydoc** module no longer displays the `self` parameter for already bound methods. Instead, it aims to always display the exact current signature of the supplied callable. (Contributed by Larry Hastings in bpo-20710.)

In addition to the changes that have been made to **pydoc** directly, its handling of custom `__dir__` methods and various descriptor behaviours has also been improved substantially by the underlying changes in the `inspect` module.

As the `help()` builtin is based on **pydoc**, the above changes also affect the behaviour of `help()`.

**re**

New `fullmatch()` function and `regex.fullmatch()` method anchor the pattern at both ends of the string to match. This provides a way to be explicit about the goal of the match, which avoids a class of subtle bugs where `$` characters get lost during code changes or the addition of alternatives to an existing regular expression. (Contributed
The repr of regex objects now includes the pattern and the flags; the repr of match objects now includes the start, end, and the part of the string that matched. (Contributed by Hugo Lopes Tavares and Serhiy Storchaka in bpo-13592 and bpo-17087.)

resource

New prlimit() function, available on Linux platforms with a kernel version of 2.6.36 or later and glibc of 2.13 or later, provides the ability to query or set the resource limits for processes other than the one making the call. (Contributed by Christian Heimes in bpo-16595.)

On Linux kernel version 2.6.36 or later, there are also some new Linux specific constants: RLIMIT_MSGQUEUE, RLIMIT_NICE, RLIMIT_RTPRIO, RLIMIT_RTTIME, and RLIMIT_SIGPENDING. (Contributed by Christian Heimes in bpo-19324.)

On FreeBSD version 9 and later, there some new FreeBSD specific constants: RLIMIT_SBSIZE, RLIMIT_SWAP, and RLIMIT_NPTS. (Contributed by Claudiu Popa in bpo-19343.)

select

epoll objects now support the context management protocol. When used in a with statement, the close() method will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in bpo-16488.)

devpoll objects now have fileno() and close() methods, as well as a new attribute closed. (Contributed by Victor Stinner in bpo-
shelve

Shelf instances may now be used in with statements, and will be automatically closed at the end of the with block. (Contributed by Filip Gruszczynski in bpo-13896.)

shutil

copyfile() now raises a specific Error subclass, SameFileError, when the source and destination are the same file, which allows an application to take appropriate action on this specific error. (Contributed by Atsuo Ishimoto and Hynek Schlawack in bpo-1492704.)

smtpd

The SMTPServer and SMTPChannel classes now accept a map keyword argument which, if specified, is passed in to asynchat.async_chat as its map argument. This allows an application to avoid affecting the global socket map. (Contributed by Vinay Sajip in bpo-11959.)

smtpplib

SMTPException is now a subclass of OSError, which allows both socket level errors and SMTP protocol level errors to be caught in one try/except statement by code that only cares whether or not an error occurred. (Contributed by Ned Jackson Lovely in bpo-2118.)
socket

The socket module now supports the CAN_BCM protocol on platforms that support it. (Contributed by Brian Thorne in bpo-15359.)

Socket objects have new methods to get or set their inheritable flag, `get_inheritable()` and `set_inheritable()`.

The `socket.AF_*` and `socket.SOCK_*` constants are now enumeration values using the new `enum` module. This allows meaningful names to be printed during debugging, instead of integer “magic numbers”.

The `AF_LINK` constant is now available on BSD and OSX.

`inet_pton()` and `inet_ntop()` are now supported on Windows. (Contributed by Atsuo Ishimoto in bpo-7171.)

sqlite3

A new boolean parameter to the `connect()` function, `uri`, can be used to indicate that the `database` parameter is a uri (see the SQLite URI documentation). (Contributed by poq in bpo-13773.)

ssl

`PROTOCOL_TLSv1_1` and `PROTOCOL_TLSv1_2` (TLSv1.1 and TLSv1.2 support) have been added; support for these protocols is only available if Python is linked with OpenSSL 1.0.1 or later. (Contributed by Michele Orrù and Antoine Pitrou in bpo-16692.)

New function `create_default_context()` provides a standard
way to obtain an SSLContext whose settings are intended to be a reasonable balance between compatibility and security. These settings are more stringent than the defaults provided by the SSLContext constructor, and may be adjusted in the future, without prior deprecation, if best-practice security requirements change. The new recommended best practice for using stdlib libraries that support SSL is to use create_default_context() to obtain an SSLContext object, modify it if needed, and then pass it as the context argument of the appropriate stdlib API. (Contributed by Christian Heimes in bpo-19689.)

SSLContext method load_verify_locations() accepts a new optional argument cadata, which can be used to provide PEM or DER encoded certificates directly via strings or bytes, respectively. (Contributed by Christian Heimes in bpo-18138.)

New function get_default_verify_paths() returns a named tuple of the paths and environment variables that the set_default_verify_paths() method uses to set OpenSSL's default cafile and capath. This can be an aid in debugging default verification issues. (Contributed by Christian Heimes in bpo-18143.)

SSLContext has a new method, cert_store_stats(), that reports the number of loaded X.509 certs, X.509 CA certs, and certificate revocation lists (crls), as well as a get_ca_certs() method that returns a list of the loaded CA certificates. (Contributed by Christian Heimes in bpo-18147.)

If OpenSSL 0.9.8 or later is available, SSLContext has a new attribute verify_flags that can be used to control the certificate verification process by setting it to some combination of the new constants VERIFY_DEFAULT, VERIFY_CRL_CHECK_LEAF,
**VERIFY_CRL_CHECK_CHAIN**, or **VERIFY_X509_STRICT**. OpenSSL does not do any CRL verification by default. (Contributed by Christien Heimes in bpo-8813.)

New **SSLContext** method **load_default_certs()** loads a set of default “certificate authority” (CA) certificates from default locations, which vary according to the platform. It can be used to load both TLS web server authentication certificates (purpose=SERVER_AUTH) for a client to use to verify a server, and certificates for a server to use in verifying client certificates (purpose=CLIENT_AUTH). (Contributed by Christian Heimes in bpo-19292.)

Two new windows-only functions, **enum_certificates()** and **enum_crls()** provide the ability to retrieve certificates, certificate information, and CRLs from the Windows cert store. (Contributed by Christian Heimes in bpo-17134.)

Support for server-side SNI (Server Name Indication) using the new **ssl.SSLContext.set_servername_callback()** method. (Contributed by Daniel Black in bpo-8109.)

The dictionary returned by **SSLSocket.getpeercert()** contains additional **X509v3** extension items: **crlDistributionPoints**, **caIssuers**, and **OCSP** URIs. (Contributed by Christian Heimes in bpo-18379.)

**stat**

The **stat** module is now backed by a C implementation in _stat. A C implementation is required as most of the values aren’t standardized and are platform-dependent. (Contributed by Christian Heimes in bpo-11016.)
The module supports new `ST_MODE` flags, `S_IFDOOR`, `S_IFPORT`, and `S_IFWHT`. (Contributed by Christian Hiemes in bpo-11016.)

**struct**

New function `iter_unpack` and a new `struct.Struct.iter_unpack()` method on compiled formats provide streamed unpacking of a buffer containing repeated instances of a given format of data. (Contributed by Antoine Pitrou in bpo-17804.)

**subprocess**

`check_output()` now accepts an `input` argument that can be used to provide the contents of `stdin` for the command that is run. (Contributed by Zack Weinberg in bpo-16624.)

getstatus() and `getstatusoutput()` now work on Windows. This change was actually inadvertently made in 3.3.4. (Contributed by Tim Golden in bpo-10197.)

**sunau**

The `getparams()` method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-18901.)

`sunau.open()` now supports the context management protocol: when used in a `with` block, the `close` method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in bpo-18878.)
AU_write.setsampwidth() now supports 24 bit samples, thus adding support for writing 24 sample using the module. (Contributed by Serhiy Storchaka in bpo-19261.)

The writeframesraw() and writeframes() methods now accept any bytes-like object. (Contributed by Serhiy Storchaka in bpo-8311.)

sys

New function sys.getallocatedblocks() returns the current number of blocks allocated by the interpreter. (In CPython with the default --with-pymalloc setting, this is allocations made through the PyObject_Malloc() API.) This can be useful for tracking memory leaks, especially if automated via a test suite. (Contributed by Antoine Pitrou in bpo-13390.)

When the Python interpreter starts in interactive mode, it checks for an __interactivehook__ attribute on the sys module. If the attribute exists, its value is called with no arguments just before interactive mode is started. The check is made after the PYTHONSTARTUP file is read, so it can be set there. The site module sets it to a function that enables tab completion and history saving (in ~/.python-history) if the platform supports readline. If you do not want this (new) behavior, you can override it in PYTHONSTARTUP, sitecustomize, or usercustomize by deleting this attribute from sys (or setting it to some other callable). (Contributed by Éric Araujo and Antoine Pitrou in bpo-5845.)

tarfile

The tarfile module now supports a simple Command-Line
Interface when called as a script directly or via -m. This can be used to create and extract tarfile archives. (Contributed by Berker Peksag in bpo-13477.)

textwrap

The TextWrapper class has two new attributes/constructor arguments: max_lines, which limits the number of lines in the output, and placeholder, which is a string that will appear at the end of the output if it has been truncated because of max_lines. Building on these capabilities, a new convenience function shorten() collapses all of the whitespace in the input to single spaces and produces a single line of a given width that ends with the placeholder (by default, [...]). (Contributed by Antoine Pitrou and Serhiy Storchaka in bpo-18585 and bpo-18725.)

threading

The Thread object representing the main thread can be obtained from the new main_thread() function. In normal conditions this will be the thread from which the Python interpreter was started. (Contributed by Andrew Svetlov in bpo-18882.)

traceback

A new traceback.clear_frames() function takes a traceback object and clears the local variables in all of the frames it references, reducing the amount of memory consumed. (Contributed by Andrew Kuchling in bpo-1565525.)
A new `DynamicClassAttribute()` descriptor provides a way to define an attribute that acts normally when looked up through an instance object, but which is routed to the `class __getattr__` when looked up through the class. This allows one to have properties active on a class, and have virtual attributes on the class with the same name (see Enum for an example). (Contributed by Ethan Furman in bpo-19030.)

**urllib**

`urllib.request` now supports `data:` URLs via the `DataHandler` class. (Contributed by Mathias Panzenböck in bpo-16423.)

The http method that will be used by a `Request` class can now be specified by setting a `method` class attribute on the subclass. (Contributed by Jason R Coombs in bpo-18978.)

`Request` objects are now reusable: if the `full_url` or `data` attributes are modified, all relevant internal properties are updated. This means, for example, that it is now possible to use the same `Request` object in more than one `OpenerDirector.open()` call with different `data` arguments, or to modify a `Request`’s `url` rather than recomputing it from scratch. There is also a new `remove_header()` method that can be used to remove headers from a `Request`. (Contributed by Alexey Kachayev in bpo-16464, Daniel Wozniak in bpo-17485, and Damien Brecht and Senthil Kumaran in bpo-17272.)

`HTTPError` objects now have a `headers` attribute that provides
access to the HTTP response headers associated with the error. (Contributed by Berker Peksag in bpo-15701.)

`unittest`

The `TestCase` class has a new method, `subTest()`, that produces a context manager whose `with` block becomes a “sub-test”. This context manager allows a test method to dynamically generate subtests by, say, calling the `subTest` context manager inside a loop. A single test method can thereby produce an indefinite number of separately-identified and separately-counted tests, all of which will run even if one or more of them fail. For example:

```python
class NumbersTest(unittest.TestCase):
    def test_even(self):
        for i in range(6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will result in six subtests, each identified in the unittest verbose output with a label consisting of the variable name `i` and a particular value for that variable (i=0, i=1, etc). See Distinguishing test iterations using subtests for the full version of this example. (Contributed by Antoine Pitrou in bpo-16997.)

`unittest.main()` now accepts an iterable of test names for `defaultTest`, where previously it only accepted a single test name as a string. (Contributed by Jyrki Pulliainen in bpo-15132.)

If `SkipTest` is raised during test discovery (that is, at the module level in the test file), it is now reported as a skip instead of an error. (Contributed by Zach Ware in bpo-16935.)
discover() now sorts the discovered files to provide consistent test ordering. (Contributed by Martin Melin and Jeff Ramnani in bpo-16709.)

TestSuite now drops references to tests as soon as the test has been run, if the test is successful. On Python interpreters that do garbage collection, this allows the tests to be garbage collected if nothing else is holding a reference to the test. It is possible to override this behavior by creating a TestSuite subclass that defines a custom _removeTestAtIndex method. (Contributed by Tom Wardill, Matt McClure, and Andrew Svetlov in bpo-11798.)

A new test assertion context-manager, assertLogs(), will ensure that a given block of code emits a log message using the logging module. By default the message can come from any logger and have a priority of INFO or higher, but both the logger name and an alternative minimum logging level may be specified. The object returned by the context manager can be queried for the LogRecords and/or formatted messages that were logged. (Contributed by Antoine Pitrou in bpo-18937.)

Test discovery now works with namespace packages (Contributed by Claudiu Popa in bpo-17457.)

unittest.mock objects now inspect their specification signatures when matching calls, which means an argument can now be matched by either position or name, instead of only by position. (Contributed by Antoine Pitrou in bpo-17015.)

mock_open() objects now have readline and readlines methods. (Contributed by Toshio Kuratomi in bpo-17467.)
venv

venv now includes activation scripts for the csh and fish shells. (Contributed by Andrew Svetlov in bpo-15417.)

EnvBuilder and the create() convenience function take a new keyword argument with_pip, which defaults to False, that controls whether or not EnvBuilder ensures that pip is installed in the virtual environment. (Contributed by Nick Coghlan in bpo-19552 as part of the PEP 453 implementation.)

wave

The getparams() method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-17487.)

wave.open() now supports the context management protocol. (Contributed by Claudiu Popa in bpo-17616.)

wave can now write output to unseekable files. (Contributed by David Jones, Guilherme Polo, and Serhiy Storchaka in bpo-5202.)

The writeframesraw() and writeframes() methods now accept any bytes-like object. (Contributed by Serhiy Storchaka in bpo-8311.)

weakref

New WeakMethod class simulates weak references to bound methods. (Contributed by Antoine Pitrou in bpo-14631.)

New finalize class makes it possible to register a callback to be
invoked when an object is garbage collected, without needing to carefully manage the lifecycle of the weak reference itself. (Contributed by Richard Oudkerk in bpo-15528.)

The callback, if any, associated with a ref is now exposed via the __callback__ attribute. (Contributed by Mark Dickinson in bpo-17643.)

xml.etree

A new parser, XMLPullParser, allows a non-blocking applications to parse XML documents. An example can be seen at Pull API for non-blocking parsing. (Contributed by Antoine Pitrou in bpo-17741.)

The xml.etree.ElementTree tostring() and tostringlist() functions, and the ElementTree write() method, now have a short_empty_elements keyword-only parameter providing control over whether elements with no content are written in abbreviated (<tag />) or expanded (<tag></tag>) form. (Contributed by Ariel Poliak and Serhiy Storchaka in bpo-14377.)

zipfile

The writepy() method of the PyZipFile class has a new filterfunc option that can be used to control which directories and files are added to the archive. For example, this could be used to exclude test files from the archive. (Contributed by Christian Tismer in bpo-19274.)

The allowZip64 parameter to ZipFile and PyZipFile is now True by default. (Contributed by William Mallard in bpo-17201.)
CPython Implementation Changes

PEP 445: Customization of CPython Memory Allocators

**PEP 445** adds new C level interfaces to customize memory allocation in the CPython interpreter.

**See also:**

**PEP 445** – Add new APIs to customize Python memory allocators
  PEP written and implemented by Victor Stinner.

PEP 442: Safe Object Finalization

**PEP 442** removes the current limitations and quirks of object finalization in CPython. With it, objects with `__del__()` methods, as well as generators with `finally` clauses, can be finalized when they are part of a reference cycle.

As part of this change, module globals are no longer forcibly set to `None` during interpreter shutdown in most cases, instead relying on the normal operation of the cyclic garbage collector. This avoids a whole class of interpreter-shutdown-time errors, usually involving `__del__` methods, that have plagued Python since the cyclic GC was first introduced.

**See also:**

**PEP 442** – Safe object finalization
PEP 456: Secure and Interchangeable Hash Algorithm

PEP 456 follows up on earlier security fix work done on Python’s hash algorithm to address certain DOS attacks to which public facing APIs backed by dictionary lookups may be subject. (See bpo-14621 for the start of the current round of improvements.) The PEP unifies CPython’s hash code to make it easier for a packager to substitute a different hash algorithm, and switches Python’s default implementation to a SipHash implementation on platforms that have a 64 bit data type. Any performance differences in comparison with the older FNV algorithm are trivial.

The PEP adds additional fields to the `sys.hash_info` struct sequence to describe the hash algorithm in use by the currently executing binary. Otherwise, the PEP does not alter any existing CPython APIs.

PEP 436: Argument Clinic

“Argument Clinic” (PEP 436) is now part of the CPython build process and can be used to simplify the process of defining and maintaining accurate signatures for builtins and standard library extension modules implemented in C.

Some standard library extension modules have been converted to use Argument Clinic in Python 3.4, and `pydoc` and `inspect` have been updated accordingly.

It is expected that signature metadata for programmatic introspection
will be added to additional callables implemented in C as part of Python 3.4 maintenance releases.

**Note:** The Argument Clinic PEP is not fully up to date with the state of the implementation. This has been deemed acceptable by the release manager and core development team in this case, as Argument Clinic will not be made available as a public API for third party use in Python 3.4.

See also:

**PEP 436 – The Argument Clinic DSL**
PEP written and implemented by Larry Hastings.

Other Build and C API Changes

- The new `PyType_GetSlot()` function has been added to the stable ABI, allowing retrieval of function pointers from named type slots when using the limited API. (Contributed by Martin von Löwis in bpo-17162.)
- The new `Py_SetStandardStreamEncoding()` pre-initialization API allows applications embedding the CPython interpreter to reliably force a particular encoding and error handler for the standard streams. (Contributed by Bastien Montagne and Nick Coghlan in bpo-16129.)
- Most Python C APIs that don’t mutate string arguments are now correctly marked as accepting `const char *` rather than `char *`. (Contributed by Serhiy Storchaka in bpo-1772673.)
- A new shell version of `python-config` can be used even when a python interpreter is not available (for example, in cross compilation scenarios).
- `PyUnicode_FromFormat()` now supports width and precision
specifications for %s, %A, %U, %v, %S, and %R. (Contributed by Ysj Ray and Victor Stinner in bpo-7330.)

- New function `PyStructSequence_InitType2()` supplements the existing `PyStructSequence_InitType()` function. The difference is that it returns 0 on success and -1 on failure.
- The CPython source can now be compiled using the address sanity checking features of recent versions of GCC and clang: the false alarms in the small object allocator have been silenced. (Contributed by Dhiru Kholia in bpo-18596.)
- The Windows build now uses Address Space Layout Randomization and Data Execution Prevention. (Contributed by Christian Heimes in bpo-16632.)
- New function `PyObject_LengthHint()` is the C API equivalent of `operator.length_hint()`. (Contributed by Armin Ronacher in bpo-16148.)

Other Improvements

- The `python` command has a new option, `-I`, which causes it to run in “isolated mode”, which means that `sys.path` contains neither the script's directory nor the user's site-packages directory, and all PYTHON* environment variables are ignored (it implies both `-s` and `-E`). Other restrictions may also be applied in the future, with the goal being to isolate the execution of a script from the user's environment. This is appropriate, for example, when Python is used to run a system script. On most POSIX systems it can and should be used in the `#!` line of system scripts. (Contributed by Christian Heimes in bpo-16499.)
- Tab-completion is now enabled by default in the interactive interpreter on systems that support `readline`. History is also enabled by default, and is written to (and read from) the file `~/.python-history`. (Contributed by Antoine Pitrou and Éric Araujo in bpo-5845.)
• Invoking the Python interpreter with **--version** now outputs the version to standard output instead of standard error (**bpo-18338**). Similar changes were made to **argparse** (**bpo-18920**) and other modules that have script-like invocation capabilities (**bpo-18922**).

• The CPython Windows installer now adds **.py** to the PATHEXT variable when extensions are registered, allowing users to run a python script at the windows command prompt by just typing its name without the **.py** extension. (Contributed by Paul Moore in **bpo-18569**.)

• A new make target **coverage-report** will build python, run the test suite, and generate an HTML coverage report for the C codebase using **gcov** and **lco**.  

• The **-R** option to the **python regression test suite** now also checks for memory allocation leaks, using **sys.getallocatedblocks()**. (Contributed by Antoine Pitrou in **bpo-13390**.)

• **python -m** now works with namespace packages.

• The **stat** module is now implemented in C, which means it gets the values for its constants from the C header files, instead of having the values hard-coded in the python module as was previously the case.

• Loading multiple python modules from a single OS module (**.so, .dll**) now works correctly (previously it silently returned the first python module in the file). (Contributed by Václav Šmilauer in **bpo-16421**.)

• A new opcode, **LOAD_CLASSDEREF**, has been added to fix a bug in the loading of free variables in class bodies that could be triggered by certain uses of **__prepare__**. (Contributed by Benjamin Peterson in **bpo-17853**.)

• A number of MemoryError-related crashes were identified and fixed by Victor Stinner using his **PEP 445**-based **pyfailmalloc** tool (**bpo-18408, bpo-18520**).

• The **pyvenv** command now accepts a **--copies** option to use
copies rather than symlinks even on systems where symlinks are the default. (Contributed by Vinay Sajip in bpo-18807.)

- The `pyvenv` command also accepts a `--without-pip` option to suppress the otherwise-automatic bootstrapping of pip into the virtual environment. (Contributed by Nick Coghlan in bpo-19552 as part of the PEP 453 implementation.)

- The encoding name is now optional in the value set for the `PYTHONIOENCODING` environment variable. This makes it possible to set just the error handler, without changing the default encoding. (Contributed by Serhiy Storchaka in bpo-18818.)

- The `bz2`, `lzma`, and `gzip` module open functions now support `x` (exclusive creation) mode. (Contributed by Tim Heaney and Vajrasky Kok in bpo-19201, bpo-19222, and bpo-19223.)

**Significant Optimizations**

- The UTF-32 decoder is now 3x to 4x faster. (Contributed by Serhiy Storchaka in bpo-14625.)

- The cost of hash collisions for sets is now reduced. Each hash table probe now checks a series of consecutive, adjacent key/hash pairs before continuing to make random probes through the hash table. This exploits cache locality to make collision resolution less expensive. The collision resolution scheme can be described as a hybrid of linear probing and open addressing. The number of additional linear probes defaults to nine. This can be changed at compile-time by defining `LINEAR_PROBES` to be any value. Set `LINEAR_PROBES=0` to turn-off linear probing entirely. (Contributed by Raymond Hettinger in bpo-18771.)

- The interpreter starts about 30% faster. A couple of measures lead to the speedup. The interpreter loads fewer modules on startup, e.g. the `re`, `collections` and `locale` modules and their dependencies are no longer imported by default. The marshal module has been improved to load compiled Python code
faster. (Contributed by Antoine Pitrou, Christian Heimes and Victor Stinner in bpo-19219, bpo-19218, bpo-19209, bpo-19205 and bpo-9548.)

- **bz2.BZ2File** is now as fast or faster than the Python2 version for most cases. **lzma.LZMAFile** has also been optimized. (Contributed by Serhiy Storchaka and Nadeem Vawda in bpo-16034.)

- **random.getrandbits()** is 20%-40% faster for small integers (the most common use case). (Contributed by Serhiy Storchaka in bpo-16674.)

- By taking advantage of the new storage format for strings, pickling of strings is now significantly faster. (Contributed by Victor Stinner and Antoine Pitrou in bpo-15596.)

- A performance issue in io.FileIO.readall() has been solved. This particularly affects Windows, and significantly speeds up the case of piping significant amounts of data through subprocess. (Contributed by Richard Oudkerk in bpo-15758.)

- **html.escape()** is now 10x faster. (Contributed by Matt Bryant in bpo-18020.)

- On Windows, the native **VirtualAlloc** is now used instead of the CRT **malloc** in **obmalloc**. Artificial benchmarks show about a 3% memory savings.

- **os.urandom()** now uses a lazily-opened persistent file descriptor so as to avoid using many file descriptors when run in parallel from multiple threads. (Contributed by Antoine Pitrou in bpo-18756.)
**Deprecated**

This section covers various APIs and other features that have been deprecated in Python 3.4, and will be removed in Python 3.5 or later. In most (but not all) cases, using the deprecated APIs will produce a `DeprecationWarning` when the interpreter is run with deprecation warnings enabled (for example, by using `-Wd`).

**Deprecations in the Python API**

- As mentioned in [PEP 451: A ModuleSpec Type for the Import System](https://www.python.org/dev/peps/pep-0451/), a number of `importlib` methods and functions are deprecated: `importlib.find_loader()` is replaced by `importlib.util.find_spec()`; `importlib.machinery.PathFinder.find_module()` is replaced by `importlib.machinery.PathFinder.find_spec()`; `importlib.abc.MetaPathFinder.find_module()` is replaced by `importlib.abc.MetaPathFinder.find_spec()`; `importlib.abc.PathEntryFinder.find_loader()` and `find_module()` are replaced by `importlib.abc.PathEntryFinder.find_spec()`; all of the `xxxLoader` ABC `load_module` methods (`importlib.abc.Loaders.load_module()`, `importlib.abc.InspectLoader.load_module()`, `importlib.abc.FileLoader.load_module()`, `importlib.abc.SourceLoader.load_module()`) should no longer be implemented, instead loaders should implement an `exec_module` method (`importlib.abc.Loader.exec_module()`),
importlib.abc.InspectLoader.exec_module()
importlib.abc.SourceLoader.exec_module()) and let
the import system take care of the rest; and
importlib.abc.Loader.module_repr(),
importlib.util.module_for_loader(),
importlib.util.set_loader(), and
importlib.util.set_package() are no longer needed
because their functions are now handled automatically by the
import system.

- The imp module is pending deprecation. To keep compatibility
  with Python 2/3 code bases, the module's removal is currently not
  scheduled.
- The formatter module is pending deprecation and is slated for
  removal in Python 3.6.
- MD5 as the default digestmod for the hmac.new() function is
deprecated. Python 3.6 will require an explicit digest name or
  constructor as digestmod argument.
- The internal Netrc class in the ftplib module has been
documented as deprecated in its docstring for quite some time. It
now emits a DeprecationWarning and will be removed
completely in Python 3.5.
- The undocumented endtime argument to
  subprocess.Popen.wait() should not have been exposed
  and is hopefully not in use; it is deprecated and will mostly likely
  be removed in Python 3.5.
- The strict argument of HTMLParser is deprecated.
- The plistlib readPlist(), writePlist(),
  readPlistFromBytes(), and writePlistToBytes() functions are deprecated in favor of the corresponding new
  functions load(), dump(), loads(), and dumps(). Data() is
deprecated in favor of just using the bytes constructor.
- The sysconfig key SO is deprecated, it has been replaced by
  EXT_SUFFIX.
• The U mode accepted by various open functions is deprecated. In Python3 it does not do anything useful, and should be replaced by appropriate uses of io.TextIOWrapper (if needed) and its newline argument.

• The parser argument of xml.etree.ElementTree.iterparse() has been deprecated, as has the html argument of XMLParser(). To prepare for the removal of the latter, all arguments to XMLParser should be passed by keyword.

Deprecated Features

• Running IDLE with the -n flag (no subprocess) is deprecated. However, the feature will not be removed until bpo-18823 is resolved.

• The site module adding a “site-python” directory to sys.path, if it exists, is deprecated (bpo-19375).
Removed

Operating Systems No Longer Supported

Support for the following operating systems has been removed from the source and build tools:

- OS/2 ([bpo-16135](https://example.com/bpo-16135)).
- Windows 2000 (changeset e52df05b496a).
- Windows systems where COMSPEC points to command.com ([bpo-14470](https://example.com/bpo-14470)).
- VMS ([bpo-16136](https://example.com/bpo-16136)).

API and Feature Removals

The following obsolete and previously deprecated APIs and features have been removed:

- The unmaintained Misc/TextMate and Misc/vim directories have been removed (see the devguide for suggestions on what to use instead).
- The S0 makefile macro is removed (it was replaced by the SHLIB_SUFFIX and EXT_SUFFIX macros) ([bpo-16754](https://example.com/bpo-16754)).
- The PyThreadState.tick_counter field has been removed; its value has been meaningless since Python 3.2, when the “new GIL” was introduced ([bpo-19199](https://example.com/bpo-19199)).
- PyLoader and PyPycLoader have been removed from importlib. (Contributed by Taras Lyapun in [bpo-15641](https://example.com/bpo-15641).)
- The strict argument to HTTPConnection and HTTPSConnection has been removed. HTTP 0.9-style “Simple Responses” are no longer supported.
- The deprecated urllib.request.Request getter and setter
methods add_data, has_data, get_data, get_type, get_host, get_selector, set_proxy, get_origin_req_host, and is_unverifiable have been removed (use direct attribute access instead).

- Support for loading the deprecated TYPE_INT64 has been removed from marshal. (Contributed by Dan Riti in bpo-15480.)
- inspect.Signature: positional-only parameters are now required to have a valid name.
- object.__format__() no longer accepts non-empty format strings, it now raises a TypeError instead. Using a non-empty string has been deprecated since Python 3.2. This change has been made to prevent a situation where previously working (but incorrect) code would start failing if an object gained a __format__ method, which means that your code may now raise a TypeError if you are using an 's' format code with objects that do not have a __format__ method that handles it. See bpo-7994 for background.
- difflib.SequenceMatcher.isbjunk() and difflib.SequenceMatcher.isbpopular() were deprecated in 3.2, and have now been removed: use x in sm bjunk and x in sm bpopular, where sm is a SequenceMatcher object (bpo-13248).

### Code Cleanups

- The unused and undocumented internal Scanner class has been removed from the pydoc module.
- The private and effectively unused _gestalt module has been removed, along with the private platform functions _mac_ver_lookup, _mac_ver_gstalt, and _bcd2str, which would only have ever been called on badly broken OSX systems (see bpo-18393).
The hardcoded copies of certain `stat` constants that were included in the `tarfile` module namespace have been removed.
Porting to Python 3.4

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in ‘python’ Command Behavior

- In a posix shell, setting the PATH environment variable to an empty value is equivalent to not setting it at all. However, setting PYTHONPATH to an empty value was not equivalent to not setting it at all: setting PYTHONPATH to an empty value was equivalent to setting it to ", which leads to confusion when reasoning by analogy to how PATH works. The behavior now conforms to the posix convention for PATH.
- The [X refs, Y blocks] output of a debug (--with-pydebug) build of the CPython interpreter is now off by default. It can be re-enabled using the -X showrefcount option. (Contributed by Ezio Melotti in bpo-17323.)
- The python command and most stdlib scripts (as well as argparse) now output --version information to stdout instead of stderr (for issue list see Other Improvements above).

Changes in the Python API

- The ABCs defined in importlib.abc now either raise the appropriate exception or return a default value instead of raising NotImplementedError blindly. This will only affect code calling super() and falling through all the way to the ABCs. For compatibility, catch both NotImplementedError or the appropriate exception as needed.
- The module type now initializes the __package__ and
__loader__ attributes to None by default. To determine if these attributes were set in a backwards-compatible fashion, use e.g. `getattr(module, '__loader__', None) is not None` (bpo-17115).

- `importlib.util.module_for_loader()` now sets __loader__ and __package__ unconditionally to properly support reloading. If this is not desired then you will need to set these attributes manually. You can use `importlib.util.module_to_load()` for module management.

- Import now resets relevant attributes (e.g. __name__, __loader__, __package__, __file__, __cached__) unconditionally when reloading. Note that this restores a pre-3.3 behavior in that it means a module is re-found when re-loaded (bpo-19413).

- Frozen packages no longer set __path__ to a list containing the package name, they now set it to an empty list. The previous behavior could cause the import system to do the wrong thing on submodule imports if there was also a directory with the same name as the frozen package. The correct way to determine if a module is a package or not is to use `hasattr(module, '__path__')` (bpo-18065).

- Frozen modules no longer define a __file__ attribute. It's semantically incorrect for frozen modules to set the attribute as they are not loaded from any explicit location. If you must know that a module comes from frozen code then you can see if the module's __spec__.location is set to 'frozen', check if the loader is a subclass of `importlib.machinery.FrozenImporter`, or if Python 2 compatibility is necessary you can use `imp.is_frozen()`.

- `py_compile.compile()` now raises FileExistsError if the file path it would write to is a symlink or a non-regular file. This is to act as a warning that import will overwrite those files with a
regular file regardless of what type of file path they were originally.

- `importlib.abc.SourceLoader.get_source()` no longer raises `ImportError` when the source code being loaded triggers a `SyntaxError` or `UnicodeDecodeError`. As `ImportError` is meant to be raised only when source code cannot be found but it should, it was felt to be over-reaching/overloading of that meaning when the source code is found but improperly structured. If you were catching `ImportError` before and wish to continue to ignore syntax or decoding issues, catch all three exceptions now.

- `functools.update_wrapper()` and `functools.wraps()` now correctly set the `__wrapped__` attribute to the function being wrapped, even if that function also had its `__wrapped__` attribute set. This means `__wrapped__` attributes now correctly link a stack of decorated functions rather than every `__wrapped__` attribute in the chain referring to the innermost function. Introspection libraries that assumed the previous behaviour was intentional can use `inspect.unwrap()` to access the first function in the chain that has no `__wrapped__` attribute.

- `inspect.getfullargspec()` has been reimplemented on top of `inspect.signature()` and hence handles a much wider variety of callable objects than it did in the past. It is expected that additional builtin and extension module callables will gain signature metadata over the course of the Python 3.4 series. Code that assumes that `inspect.getfullargspec()` will fail on non-Python callables may need to be adjusted accordingly.

- `importlib.machinery.PathFinder` now passes on the current working directory to objects in `sys.path_hooks` for the empty string. This results in `sys.path_importer_cache` never containing `''`, thus iterating through `sys.path_importer_cache` based on `sys.path` will not find all keys. A module's `__file__` when imported in the current working directory will also now have an absolute path, including
when using `-m` with the interpreter (except for `__main__.__file__` when a script has been executed directly using a relative path) (Contributed by Brett Cannon in bpo-18416).

- The removal of the `strict` argument to `HTTPConnection` and `HTTPSConnection` changes the meaning of the remaining arguments if you are specifying them positionally rather than by keyword. If you've been paying attention to deprecation warnings your code should already be specifying any additional arguments via keywords.

- Strings between `from __future__ import ...` statements now always raise a `SyntaxError`. Previously if there was no leading docstring, an interstitial string would sometimes be ignored. This brings CPython into compliance with the language spec; Jython and PyPy already were. (bpo-17434).

- `ssl.SSLSocket.getpeercert()` and `ssl.SSLSocket.do_handshake()` now raise an `OSError` with `ENOTCONN` when the `SSLSocket` is not connected, instead of the previous behavior of raising an `AttributeError`. In addition, `getpeercert()` will raise a `ValueError` if the handshake has not yet been done.

- `base64.b32decode()` now raises a `binascii.Error` when the input string contains non-b32-alphabet characters, instead of a `TypeError`. This particular `TypeError` was missed when the other `TypeError`s were converted. (Contributed by Serhiy Storchaka in bpo-18011.) Note: this change was also inadvertently applied in Python 3.3.3.

- The file attribute is now automatically closed when the creating `cgi.FieldStorage` instance is garbage collected. If you were pulling the file object out separately from the `cgi.FieldStorage` instance and not keeping the instance alive, then you should either store the entire `cgi.FieldStorage` instance or read the contents of the file
before the `cgi.FieldStorage` instance is garbage collected.

- Calling `read` or `write` on a closed SSL socket now raises an informative `ValueError` rather than the previous more mysterious `AttributeError` (bpo-9177).
- `slice.indices()` no longer produces an `OverflowError` for huge values. As a consequence of this fix, `slice.indices()` now raises a `ValueError` if given a negative length; previously it returned nonsense values (bpo-14794).
- The `complex` constructor, unlike the `cmath` functions, was incorrectly accepting `float` values if an object's `__complex__` special method returned one. This now raises a `TypeError` (bpo-16290).
- The `int` constructor in 3.2 and 3.3 erroneously accepts `float` values for the `base` parameter. It is unlikely anyone was doing this, but if so, it will now raise a `TypeError` (bpo-16772).
- Defaults for keyword-only arguments are now evaluated after defaults for regular keyword arguments, instead of before. Hopefully no one wrote any code that depends on the previous buggy behavior (bpo-16967).
- Stale thread states are now cleared after `fork()`. This may cause some system resources to be released that previously were incorrectly kept perpetually alive (for example, database connections kept in thread-local storage). (bpo-17094.)
- Parameter names in `__annotations__` dicts are now mangled properly, similarly to `__kwdefaults__`. (Contributed by Yury Selivanov in bpo-20625.)
- `hashlib.hash.name` now always returns the identifier in lower case. Previously some built-in hashes had uppercase names, but now that it is a formal public interface the naming has been made consistent (bpo-18532).
- Because `unittest.TestSuite` now drops references to tests after they are run, test harnesses that re-use a `TestSuite` to re-run a set of tests may fail. Test suites should not be re-used in this
fashion since it means state is retained between test runs, breaking the test isolation that unittest is designed to provide. However, if the lack of isolation is considered acceptable, the old behavior can be restored by creating a TestSuite subclass that defines a `_removeTestAtIndex` method that does nothing (see TestSuite.__iter__(bpo-11798)).

- unittest now uses argparse for command line parsing. There are certain invalid command forms that used to work that are no longer allowed; in theory this should not cause backward compatibility issues since the disallowed command forms didn’t make any sense and are unlikely to be in use.
- The `re.split()`, `re.findall()`, and `re.sub()` functions, and the `group()` and `groups()` methods of match objects now always return a bytes object when the string to be matched is a bytes-like object. Previously the return type matched the input type, so if your code was depending on the return value being, say, a bytearray, you will need to change your code.
- audioop functions now raise an error immediately if passed string input, instead of failing randomly later on (bpo-16685).
- The new `convert_charrefs` argument to HTMLParser currently defaults to False for backward compatibility, but will eventually be changed to default to True. It is recommended that you add this keyword, with the appropriate value, to any HTMLParser calls in your code (bpo-13633).
- Since the `digestmod` argument to the hmac.new() function will in the future have no default, all calls to hmac.new() should be changed to explicitly specify a digestmod (bpo-17276).
- Calling `sysconfig.get_config_var()` with the SO key, or looking SO up in the results of a call to `sysconfig.get_config_vars()` is deprecated. This key should be replaced by EXT_SUFFIX or SHLIB_SUFFIX, depending on the context (bpo-19555).
- Any calls to open functions that specify U should be modified. U
is ineffective in Python3 and will eventually raise an error if used. Depending on the function, the equivalent of its old Python2 behavior can be achieved using either a newline argument, or if necessary by wrapping the stream in TextIOWrapper to use its newline argument (bpo-15204).

- If you use pyvenv in a script and desire that pip not be installed, you must add --without-pip to your command invocation.
- The default behavior of json.dump() and json.dumps() when an indent is specified has changed: it no longer produces trailing spaces after the item separating commas at the ends of lines. This will matter only if you have tests that are doing white-space-sensitive comparisons of such output (bpo-16333).
- doctest now looks for doctests in extension module __doc__ strings, so if your doctest test discovery includes extension modules that have things that look like doctests in them you may see test failures you've never seen before when running your tests (bpo-3158).
- The collections.abc module has been slightly refactored as part of the Python startup improvements. As a consequence of this, it is no longer the case that importing collections automatically imports collections.abc. If your program depended on the (undocumented) implicit import, you will need to add an explicit import collections.abc (bpo-20784).

Changes in the C API

- PyEval_EvalFrameEx(), PyObject_Repr(), and PyObject_Str(), along with some other internal C APIs, now include a debugging assertion that ensures they are not used in situations where they may silently discard a currently active exception. In cases where discarding the active exception is expected and desired (for example, because it has already been saved locally with PyErr_Fetch() or is being deliberately
replaced with a different exception), an explicit `PyErr_Clear()` call will be needed to avoid triggering the assertion when invoking these operations (directly or indirectly) and running against a version of Python that is compiled with assertions enabled.

- `PyErr_SetImportError()` now sets `TypeError` when its `msg` argument is not set. Previously only `NULL` was returned with no exception set.
- The result of the `PyOS_ReadlineFunctionPointer` callback must now be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or `NULL` if an error occurred, instead of a string allocated by `PyMem_Malloc()` or `PyMem_Realloc()` (bpo-16742)
- `PyThread_set_key_value()` now always set the value. In Python 3.3, the function did nothing if the key already exists (if the current value is a non-NULL pointer).
- The `f_tstate` (thread state) field of the `PyFrameObject` structure has been removed to fix a bug: see bpo-14432 for the rationale.
Changed in 3.4.3

PEP 476: Enabling certificate verification by default for stdlib http clients

http.client and modules which use it, such as urllib.request and xmlrpc.client, will now verify that the server presents a certificate which is signed by a CA in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications.

For applications which require the old previous behavior, they can pass an alternate context:

```python
import urllib.request
import ssl

# This disables all verification
context = ssl._create_unverified_context()

# This allows using a specific certificate for the host to be in the trust store
context = ssl.create_default_context(cafile="/path/to/file.crt")

urllib.request.urlopen("https://invalid-cert", context=context)
```
What’s New In Python 3.3

This article explains the new features in Python 3.3, compared to 3.2. Python 3.3 was released on September 29, 2012. For full details, see the changelog.

See also:  PEP 398 - Python 3.3 Release Schedule
Summary – Release highlights

New syntax features:

- New `yield from` expression for generator delegation.
- The `u'unicode'` syntax is accepted again for `str` objects.

New library modules:

- `faulthandler` (helps debugging low-level crashes)
- `ipaddress` (high-level objects representing IP addresses and masks)
- `lzma` (compress data using the XZ / LZMA algorithm)
- `unittest.mock` (replace parts of your system under test with mock objects)
- `venv` (Python virtual environments, as in the popular `virtualenv` package)

New built-in features:

- Reworked `I/O exception hierarchy`.

Implementation improvements:

- Rewritten `import machinery` based on `importlib`.
- More compact `unicode strings`.
- More compact `attribute dictionaries`.

Significantly Improved Library Modules:

- C Accelerator for the `decimal` module.
- Better unicode handling in the `email` module (provisional).

Security improvements:
- Hash randomization is switched on by default.

Please read on for a comprehensive list of user-facing changes.
Virtual environments help create separate Python setups while sharing a system-wide base install, for ease of maintenance. Virtual environments have their own set of private site packages (i.e. locally-installed libraries), and are optionally segregated from the system-wide site packages. Their concept and implementation are inspired by the popular virtualenv third-party package, but benefit from tighter integration with the interpreter core.

This PEP adds the venv module for programmatic access, and the pyvenv script for command-line access and administration. The Python interpreter checks for a pyvenv.cfg, file whose existence signals the base of a virtual environment's directory tree.

See also:

PEP 405 - Python Virtual Environments
PEP written by Carl Meyer; implementation by Carl Meyer and Vinay Sajip
PEP 420: Implicit Namespace Packages

Native support for package directories that don't require __init__.py marker files and can automatically span multiple path segments (inspired by various third party approaches to namespace packages, as described in PEP 420)

See also:

PEP 420 - Implicit Namespace Packages
PEP written by Eric V. Smith; implementation by Eric V. Smith and Barry Warsaw
PEP 3118: New memoryview implementation and buffer protocol documentation

The implementation of PEP 3118 has been significantly improved.

The new memoryview implementation comprehensively fixes all ownership and lifetime issues of dynamically allocated fields in the Py_buffer struct that led to multiple crash reports. Additionally, several functions that crashed or returned incorrect results for non-contiguous or multi-dimensional input have been fixed.

The memoryview object now has a PEP-3118 compliant getbufferproc() that checks the consumer’s request type. Many new features have been added, most of them work in full generality for non-contiguous arrays and arrays with suboffsets.

The documentation has been updated, clearly spelling out responsibilities for both exporters and consumers. Buffer request flags are grouped into basic and compound flags. The memory layout of non-contiguous and multi-dimensional NumPy-style arrays is explained.

Features

- All native single character format specifiers in struct module syntax (optionally prefixed with ‘@’) are now supported.
- With some restrictions, the cast() method allows changing of format and shape of C-contiguous arrays.
- Multi-dimensional list representations are supported for any array type.
Multi-dimensional comparisons are supported for any array type.
One-dimensional memoryviews of hashable (read-only) types with formats B, b or c are now hashable. (Contributed by Antoine Pitrou in bpo-13411.)
Arbitrary slicing of any 1-D arrays type is supported. For example, it is now possible to reverse a memoryview in O(1) by using a negative step.

API changes

The maximum number of dimensions is officially limited to 64.
The representation of empty shape, strides and suboffsets is now an empty tuple instead of None.
Accessing a memoryview element with format ‘B’ (unsigned bytes) now returns an integer (in accordance with the struct module syntax). For returning a bytes object the view must be cast to ‘c’ first.
memoryview comparisons now use the logical structure of the operands and compare all array elements by value. All format strings in struct module syntax are supported. Views with unrecognised format strings are still permitted, but will always compare as unequal, regardless of view contents.
For further changes see Build and C API Changes and Porting C code.

(Contributed by Stefan Krah in bpo-10181.)

See also:  PEP 3118 - Revising the Buffer Protocol
PEP 393: Flexible String Representation

The Unicode string type is changed to support multiple internal representations, depending on the character with the largest Unicode ordinal (1, 2, or 4 bytes) in the represented string. This allows a space-efficient representation in common cases, but gives access to full UCS-4 on all systems. For compatibility with existing APIs, several representations may exist in parallel; over time, this compatibility should be phased out.

On the Python side, there should be no downside to this change.

On the C API side, PEP 393 is fully backward compatible. The legacy API should remain available at least five years. Applications using the legacy API will not fully benefit of the memory reduction, or - worse - may use a bit more memory, because Python may have to maintain two versions of each string (in the legacy format and in the new efficient storage).

Functionality

Changes introduced by PEP 393 are the following:

- Python now always supports the full range of Unicode code points, including non-BMP ones (i.e. from U+0000 to U+1FFFF). The distinction between narrow and wide builds no longer exists and Python now behaves like a wide build, even under Windows.
- With the death of narrow builds, the problems specific to narrow builds have also been fixed, for example:
  - `len()` now always returns 1 for non-BMP characters, so `len('\U0010FFFF') == 1`;
  - surrogate pairs are not recombined in string literals, so
indexing or slicing non-BMP characters returns the expected value, so '\U0010FFFF'[0] now returns '\U0010FFFF' and not '\uDBFF';

all other functions in the standard library now correctly handle non-BMP code points.

- The value of \texttt{sys.maxunicode} is now always 1114111 (0x10FFFF in hexadecimal). The \texttt{PyUnicode_GetMax()} function still returns either 0xFFFF or 0x10FFFF for backward compatibility, and it should not be used with the new Unicode API (see \texttt{bpo-13054}).
- The \texttt{./configure} flag \texttt{--with-wide-unicode} has been removed.

Performance and resource usage

The storage of Unicode strings now depends on the highest code point in the string:

- pure ASCII and Latin1 strings (U+0000-U+00FF) use 1 byte per code point;
- BMP strings (U+0000-U+FFFF) use 2 bytes per code point;
- non-BMP strings (U+10000-U+10FFFF) use 4 bytes per code point.

The net effect is that for most applications, memory usage of string storage should decrease significantly - especially compared to former wide unicode builds - as, in many cases, strings will be pure ASCII even in international contexts (because many strings store non-human language data, such as XML fragments, HTTP headers, JSON-encoded data, etc.). We also hope that it will, for the same reasons, increase CPU cache efficiency on non-trivial applications. The memory usage of Python 3.3 is two to three times smaller than Python 3.2, and
a little bit better than Python 2.7, on a Django benchmark (see the PEP for details).

See also:

PEP 393 - Flexible String Representation
PEP written by Martin von Löwis; implementation by Torsten Becker and Martin von Löwis.
PEP 397: Python Launcher for Windows

The Python 3.3 Windows installer now includes a `py` launcher application that can be used to launch Python applications in a version independent fashion.

This launcher is invoked implicitly when double-clicking `*.py` files. If only a single Python version is installed on the system, that version will be used to run the file. If multiple versions are installed, the most recent version is used by default, but this can be overridden by including a Unix-style “shebang line” in the Python script.

The launcher can also be used explicitly from the command line as the `py` application. Running `py` follows the same version selection rules as implicitly launching scripts, but a more specific version can be selected by passing appropriate arguments (such as `-3` to request Python 3 when Python 2 is also installed, or `-2.6` to specifically request an earlier Python version when a more recent version is installed).

In addition to the launcher, the Windows installer now includes an option to add the newly installed Python to the system PATH. (Contributed by Brian Curtin in bpo-3561.)

See also:

**PEP 397 - Python Launcher for Windows**

PEP written by Mark Hammond and Martin v. Löwis; implementation by Vinay Sajip.

Launcher documentation: [Python Launcher for Windows](#)
Installer PATH modification: Finding the Python executable
PEP 3151: Reworking the OS and IO exception hierarchy

The hierarchy of exceptions raised by operating system errors is now both simplified and finer-grained.

You don't have to worry anymore about choosing the appropriate exception type between OSError, IOError, EnvironmentError, WindowsError, mmap.error, socket.error or select.error. All these exception types are now only one: OSError. The other names are kept as aliases for compatibility reasons.

Also, it is now easier to catch a specific error condition. Instead of inspecting the errno attribute (or args[0]) for a particular constant from the errno module, you can catch the adequate OSError subclass. The available subclasses are the following:

- BlockingIOError
- ChildProcessError
- ConnectionError
- FileExistsError
- FileNotFoundError
- InterruptedError
- IsADirectoryError
- NotADirectoryError
- PermissionError
- ProcessLookupError
- TimeoutError

And the ConnectionError itself has finer-grained subclasses:
• BrokenPipeError
• ConnectionAbortedError
• ConnectionRefusedError
• ConnectionResetError

Thanks to the new exceptions, common usages of the errno can now be avoided. For example, the following code written for Python 3.2:

```python
from errno import ENOENT, EACCES, EPERM

try:
    with open("document.txt") as f:
        content = f.read()
except IOError as err:
    if err.errno == ENOENT:
        print("document.txt file is missing")
    elif err.errno in (EACCES, EPERM):
        print("You are not allowed to read document.
    else:
        raise
```

can now be written without the errno import and without manual inspection of exception attributes:

```python
try:
    with open("document.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("document.txt file is missing")
except PermissionError:
    print("You are not allowed to read document.txt"
```

See also:
PEP 3151 - Reworking the OS and IO Exception Hierarchy
PEP written and implemented by Antoine Pitrou
PEP 380: Syntax for Delegating to a Subgenerator

PEP 380 adds the `yield from` expression, allowing a generator to delegate part of its operations to another generator. This allows a section of code containing `yield` to be factored out and placed in another generator. Additionally, the subgenerator is allowed to return with a value, and the value is made available to the delegating generator.

While designed primarily for use in delegating to a subgenerator, the `yield from` expression actually allows delegation to arbitrary subiterators.

For simple iterators, `yield from iterable` is essentially just a shortened form of `for item in iterable: yield item`:

```python
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

However, unlike an ordinary loop, `yield from` allows subgenerators to receive sent and thrown values directly from the calling scope, and return a final value to the outer generator:

```python
>>> def accumulate():
...     tally = 0
...     while 1:
...         next = yield
```
The main principle driving this change is to allow even generators that are designed to be used with the `send` and `throw` methods to be split into multiple subgenerators as easily as a single large function can be split into multiple subfunctions.

**See also:**

**PEP 380 - Syntax for Delegating to a Subgenerator**

PEP written by Greg Ewing; implementation by Greg Ewing, integrated into 3.3 by Renaud Blanch, Ryan Kelly and Nick Coghlan; documentation by Zbigniew Jędrzejewski-Szmek and Nick Coghlan
PEP 409: Suppressing exception context

PEP 409 introduces new syntax that allows the display of the chained exception context to be disabled. This allows cleaner error messages in applications that convert between exception types:

```python
>>> class D:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr) from None

>>> D({}).x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

Without the `from None` suffix to suppress the cause, the original exception would be displayed by default:

```python
>>> class C:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr)

>>> C({}).x
```
Traceback (most recent call last):
  File "<stdin>", line 6, in __getattr__
KeyError: 'x'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x

No debugging capability is lost, as the original exception context remains available if needed (for example, if an intervening library has incorrectly suppressed valuable underlying details):

```python
>>> try:
...   D({}).x
... except AttributeError as exc:
...   print(repr(exc.__context__))
... KeyError('x',)
```

See also:

**PEP 409** - Suppressing exception context

PEP written by Ethan Furman; implemented by Ethan Furman and Nick Coghlan.
PEP 414: Explicit Unicode literals

To ease the transition from Python 2 for Unicode aware Python applications that make heavy use of Unicode literals, Python 3.3 once again supports the “u” prefix for string literals. This prefix has no semantic significance in Python 3, it is provided solely to reduce the number of purely mechanical changes in migrating to Python 3, making it easier for developers to focus on the more significant semantic changes (such as the stricter default separation of binary and text data).

See also:

PEP 414 - Explicit Unicode literals
PEP written by Armin Ronacher.
PEP 3155: Qualified name for classes and functions

Functions and class objects have a new `__qualname__` attribute representing the “path” from the module top-level to their definition. For global functions and classes, this is the same as `__name__`. For other functions and classes, it provides better information about where they were actually defined, and how they might be accessible from the global scope.

Example with (non-bound) methods:

```python
>>> class C:
...     def meth(self):
...         pass
>>> C.meth.__name__
'meth'
>>> C.meth.__qualname__
'C.meth'
```

Example with nested classes:

```python
>>> class C:
...     class D:
...         def meth(self):
...             pass
...     ...

>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
```
Example with nested functions:

```python
>>> def outer():
...     def inner():
...         pass
...     return inner
...
>>> outer()().__name__
'inner'
>>> outer()().__qualname__
'outer.<locals>.inner'
```

The string representation of those objects is also changed to include the new, more precise information:

```python
>>> str(C.D)
"<class '__main__.C.D'>"
>>> str(C.D.meth)
'<function C.D.meth at 0x7f46b9fe31e0>'
```

See also:

**PEP 3155 - Qualified name for classes and functions**

PEP written and implemented by Antoine Pitrou.
PEP 412: Key-Sharing Dictionary

Dictionaries used for the storage of objects’ attributes are now able to share part of their internal storage between each other (namely, the part which stores the keys and their respective hashes). This reduces the memory consumption of programs creating many instances of non-built-in types.

See also:

PEP 412 - Key-Sharing Dictionary
  PEP written and implemented by Mark Shannon.
PEP 362: Function Signature Object

A new function `inspect.signature()` makes introspection of python callables easy and straightforward. A broad range of callables is supported: python functions, decorated or not, classes, and `functools.partial()` objects. New classes `inspect.Signature`, `inspect.Parameter` and `inspect.BoundArguments` hold information about the call signatures, such as, annotations, default values, parameters kinds, and bound arguments, which considerably simplifies writing decorators and any code that validates or amends calling signatures or arguments.

See also:

**PEP 362**: - Function Signature Object

PEP written by Brett Cannon, Yury Selivanov, Larry Hastings, Jiwon Seo; implemented by Yury Selivanov.
PEP 421: Adding sys.implementation

A new attribute on the sys module exposes details specific to the implementation of the currently running interpreter. The initial set of attributes on sys.implementation are name, version, hexversion, and cache_tag.

The intention of sys.implementation is to consolidate into one namespace the implementation-specific data used by the standard library. This allows different Python implementations to share a single standard library code base much more easily. In its initial state, sys.implementation holds only a small portion of the implementation-specific data. Over time that ratio will shift in order to make the standard library more portable.

One example of improved standard library portability is cache_tag. As of Python 3.3, sys.implementation.cache_tag is used by importlib to support PEP 3147 compliance. Any Python implementation that uses importlib for its built-in import system may use cache_tag to control the caching behavior for modules.

SimpleNamespace

The implementation of sys.implementation also introduces a new type to Python: types.SimpleNamespace. In contrast to a mapping-based namespace, like dict, SimpleNamespace is attribute-based, like object. However, unlike object, SimpleNamespace instances are writable. This means that you can add, remove, and modify the namespace through normal attribute access.
See also:

PEP 421 - Adding sys.implementation

PEP written and implemented by Eric Snow.
Using importlib as the Implementation of Import

**bpo-2377** - Replace `__import__` w/ importlib.`__import__` **bpo-13959** - Re-implement parts of `imp` in pure Python **bpo-14605** - Make import machinery explicit **bpo-14646** - Require loaders set `__loader__` and `__package__`

The `__import__()` function is now powered by `importlib.__import__()`. This work leads to the completion of “phase 2” of PEP 302. There are multiple benefits to this change. First, it has allowed for more of the machinery powering import to be exposed instead of being implicit and hidden within the C code. It also provides a single implementation for all Python VMs supporting Python 3.3 to use, helping to end any VM-specific deviations in import semantics. And finally it eases the maintenance of import, allowing for future growth to occur.

For the common user, there should be no visible change in semantics. For those whose code currently manipulates import or calls import programmatically, the code changes that might possibly be required are covered in the Porting Python code section of this document.

**New APIs**

One of the large benefits of this work is the exposure of what goes into making the import statement work. That means the various importers that were once implicit are now fully exposed as part of the `importlib` package.

The abstract base classes defined in `importlib.abc` have been
expanded to properly delineate between meta path finders and path entry finders by introducing `importlib.abc.MetaPathFinder` and `importlib.abc.PathEntryFinder`, respectively. The old ABC of `importlib.abc.Finder` is now only provided for backwards-compatibility and does not enforce any method requirements.

In terms of finders, `importlib.machinery.FileFinder` exposes the mechanism used to search for source and bytecode files of a module. Previously this class was an implicit member of `sys.path_hooks`.

For loaders, the new abstract base class `importlib.abc.FileLoader` helps write a loader that uses the file system as the storage mechanism for a module's code. The loader for source files (`importlib.machinery.SourceFileLoader`), sourceless bytecode files (`importlib.machinery.SourcelessFileLoader`), and extension modules (`importlib.machinery.ExtensionFileLoader`) are now available for direct use.

`ImportError` now has `name` and `path` attributes which are set when there is relevant data to provide. The message for failed imports will also provide the full name of the module now instead of just the tail end of the module's name.

The `importlib.invalidate_caches()` function will now call the method with the same name on all finders cached in `sys.path_importer_cache` to help clean up any stored state as necessary.

Visible Changes
For potential required changes to code, see the Porting Python code section.

Beyond the expanse of what importlib now exposes, there are other visible changes to import. The biggest is that sys.meta_path and sys.path_hooks now store all of the meta path finders and path entry hooks used by import. Previously the finders were implicit and hidden within the C code of import instead of being directly exposed. This means that one can now easily remove or change the order of the various finders to fit one’s needs.

Another change is that all modules have a __loader__ attribute, storing the loader used to create the module. PEP 302 has been updated to make this attribute mandatory for loaders to implement, so in the future once 3rd-party loaders have been updated people will be able to rely on the existence of the attribute. Until such time, though, import is setting the module post-load.

Loaders are also now expected to set the __package__ attribute from PEP 366. Once again, import itself is already setting this on all loaders from importlib and import itself is setting the attribute post-load.

None is now inserted into sys.path_importer_cache when no finder can be found on sys.path_hooks. Since imp.NullImporter is not directly exposed on sys.path_hooks it could no longer be relied upon to always be available to use as a value representing no finder found.

All other changes relate to semantic changes which should be taken into consideration when updating code for Python 3.3, and thus should be read about in the Porting Python code section of this document.
(Implementation by Brett Cannon)
Other Language Changes

Some smaller changes made to the core Python language are:

- Added support for Unicode name aliases and named sequences. Both `unicodedata.lookup()` and `'\N{...}'` now resolve name aliases, and `unicodedata.lookup()` resolves named sequences too.

  (Contributed by Ezio Melotti in `bpo-12753`.)

- Unicode database updated to UCD version 6.1.0

- Equality comparisons on `range()` objects now return a result reflecting the equality of the underlying sequences generated by those range objects. (`bpo-13201`)

- The `count()`, `find()`, `rfind()`, `index()` and `rindex()` methods of `bytes` and `bytearray` objects now accept an integer between 0 and 255 as their first argument.

  (Contributed by Petri Lehtinen in `bpo-12170`.)

- The `rjust()`, `ljust()`, and `center()` methods of `bytes` and `bytearray` now accept a `bytearray` for the `fill` argument.

  (Contributed by Petri Lehtinen in `bpo-12380`.)

- New methods have been added to `list` and `bytearray`: `copy()` and `clear()` (`bpo-10516`). Consequently, `MutableSequence` now also defines a clear() method (`bpo-11388`).

- Raw bytes literals can now be written `rb"..."` as well as
dict.setdefault() now does only one lookup for the given key, making it atomic when used with built-in types.

(Contributed by Filip Gruszczynski in bpo-13521.)

The error messages produced when a function call does not match the function signature have been significantly improved.

(Contributed by Benjamin Peterson.)
A Finer-Grained Import Lock

Previous versions of CPython have always relied on a global import lock. This led to unexpected annoyances, such as deadlocks when importing a module would trigger code execution in a different thread as a side-effect. Clumsy workarounds were sometimes employed, such as the `PyImport_ImportModuleNoBlock()` C API function.

In Python 3.3, importing a module takes a per-module lock. This correctly serializes importation of a given module from multiple threads (preventing the exposure of incompletely initialized modules), while eliminating the aforementioned annoyances.

(Contributed by Antoine Pitrou in bpo-9260.)
Built-in functions and types

- **open()**: gets a new *opener* parameter: the underlying file descriptor for the file object is then obtained by calling *opener* with *(file, flags)*. It can be used to use custom flags like `os.O_CLOEXEC` for example. The `'x'` mode was added: open for exclusive creation, failing if the file already exists.

- **print()**: added the *flush* keyword argument. If the *flush* keyword argument is true, the stream is forcibly flushed.

- **hash()**: hash randomization is enabled by default, see `object.__hash__() and PYTHONHASHSEED`.

- The `str` type gets a new *casefold() method*: return a casefolded copy of the string, casefolded strings may be used for caseless matching. For example, `'ß'.casefold()` returns `'ss'`.

- The sequence documentation has been substantially rewritten to better explain the binary/text sequence distinction and to provide specific documentation sections for the individual built-in sequence types *(bpo-4966)*.
New Modules

faulthandler

This new debug module **fau** thandler contains functions to dump Python tracebacks explicitly, on a fault (a crash like a segmentation fault), after a timeout, or on a user signal. Call **fau** thandler.enable() to install fault handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS, and SIGILL signals. You can also enable them at startup by setting the **PYTHONFAULTHANDLER** environment variable or by using **-X faulthandler** command line option.

Example of a segmentation fault on Linux:

```bash
$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault
```

Current thread 0x00007fb899f39700:
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault

ipaddress

The new **ipaddress** module provides tools for creating and manipulating objects representing IPv4 and IPv6 addresses, networks and interfaces (i.e. an IP address associated with a specific IP subnet).

(Contributed by Google and Peter Moody in PEP 3144.)
Izma

The newly-added *lzma* module provides data compression and decompression using the LZMA algorithm, including support for the `.xz` and `.lzma` file formats.

(Contributed by Nadeem Vawda and Per Øyvind Karlsen in *bpo-6715*. )
Improved Modules

abc

Improved support for abstract base classes containing descriptors composed with abstract methods. The recommended approach to declaring abstract descriptors is now to provide __isabstractmethod__ as a dynamically updated property. The built-in descriptors have been updated accordingly.

- **abc.abstractproperty** has been deprecated, use `property` with `abc.abstractmethod()` instead.
- **abc.abstractclassmethod** has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
- **abc.abstractstaticmethod** has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.

(Contributed by Darren Dale in bpo-11610.)

**abc.ABCMeta.register()** now returns the registered subclass, which means it can now be used as a class decorator (bpo-10868).

array

The **array** module supports the `long` `long` type using `q` and `Q` type codes.

(Contributed by Oren Tirosh and Hirokazu Yamamoto in bpo-1172711.)

base64

ASCII-only Unicode strings are now accepted by the decoding
functions of the base64 modern interface. For example, base64.b64decode('YWJj') returns b'abc'. (Contributed by Catalin Iacob in bpo-13641.)

binascii

In addition to the binary objects they normally accept, the a2b_ functions now all also accept ASCII-only strings as input. (Contributed by Antoine Pitrou in bpo-13637.)

bz2

The bz2 module has been rewritten from scratch. In the process, several new features have been added:

- New bz2.open() function: open a bzip2-compressed file in binary or text mode.
- bz2.BZ2File can now read from and write to arbitrary file-like objects, by means of its constructor’s fileobj argument.
  (Contributed by Nadeem Vawda in bpo-5863.)
- bz2.BZ2File and bz2.decompress() can now decompress multi-stream inputs (such as those produced by the pbzip2 tool). bz2.BZ2File can now also be used to create this type of file, using the 'a' (append) mode.
  (Contributed by Nir Aides in bpo-1625.)
- bz2.BZ2File now implements all of the io.BufferedIOBase API, except for the detach() and truncate() methods.
The mbcs codec has been rewritten to handle correctly replace and ignore error handlers on all Windows versions. The mbcs codec now supports all error handlers, instead of only replace to encode and ignore to decode.

A new Windows-only codec has been added: cp65001 (bpo-13216). It is the Windows code page 65001 (Windows UTF-8, CP_UTF8). For example, it is used by sys.stdout if the console output code page is set to cp65001 (e.g., using chcp 65001 command).

Multibyte CJK decoders now resynchronize faster. They only ignore the first byte of an invalid byte sequence. For example, b'\xff\n'.decode('gb2312', 'replace') now returns a \n after the replacement character.

(bpo-12016)

Incremental CJK codec encoders are no longer reset at each call to their encode() methods. For example:

```python
>>> import codecs
>>> encoder = codecs.getincrementalencoder('hz')('strict')
>>> b''.join(encoder.encode(x) for x in '\u52ff\u65bd\u4eba\n')
'b'^{NpJ)l6HK!#~} Bye.'
```

This example gives b'^{Np~}~{J}~{16~}~{HK~}~{!#~} Bye.' with older Python versions.

(bpo-12100)
The `unicode_internal` codec has been deprecated.

collections

Addition of a new `ChainMap` class to allow treating a number of mappings as a single unit. (Written by Raymond Hettinger for `bpo-11089`, made public in `bpo-11297`.)

The abstract base classes have been moved in a new `collections.abc` module, to better differentiate between the abstract and the concrete collections classes. Aliases for ABCs are still present in the `collections` module to preserve existing imports. (`bpo-11085`)

The `Counter` class now supports the unary `+` and `-` operators, as well as the in-place operators `+=`, `-=` , `|=`, and `&=`. (Contributed by Raymond Hettinger in `bpo-13121`.)

contextlib

`ExitStack` now provides a solid foundation for programmatic manipulation of context managers and similar cleanup functionality. Unlike the previous `contextlib.nested` API (which was deprecated and removed), the new API is designed to work correctly regardless of whether context managers acquire their resources in their `__init__` method (for example, file objects) or in their `__enter__` method (for example, synchronisation objects from the `threading` module).

(`bpo-13585`)
crypt

Addition of salt and modular crypt format (hashing method) and the mksalt() function to the crypt module.

(bpo-10924)

curses

- If the curses module is linked to the ncursesw library, use Unicode functions when Unicode strings or characters are passed (e.g. waddwstr()), and bytes functions otherwise (e.g. waddstr()).
- Use the locale encoding instead of utf-8 to encode Unicode strings.
- curses.window has a new curses.window.encoding attribute.
- The curses.window class has a new get_wch() method to get a wide character
- The curses module has a new unget_wch() function to push a wide character so the next get_wch() will return it

(Contributed by Iñigo Serna in bpo-6755.)

datetime

- Equality comparisons between naive and aware datetime instances now return False instead of raising TypeError (bpo-15006).
- New datetime.datetime.timestamp() method: Return POSIX timestamp corresponding to the datetime instance.
- The datetime.datetime.strftime() method supports
- Formatting years older than 1000.
- The `datetime.datetime.astimezone()` method can now be called without arguments to convert datetime instance to the system timezone.

**decimal**

bpo-7652 - integrate fast native decimal arithmetic.
   C-module and libmpdec written by Stefan Krah.

The new C version of the decimal module integrates the high speed libmpdec library for arbitrary precision correctly-rounded decimal floating point arithmetic. libmpdec conforms to IBM’s General Decimal Arithmetic Specification.

Performance gains range from 10x for database applications to 100x for numerically intensive applications. These numbers are expected gains for standard precisions used in decimal floating point arithmetic. Since the precision is user configurable, the exact figures may vary. For example, in integer bignum arithmetic the differences can be significantly higher.

The following table is meant as an illustration. Benchmarks are available at [http://www.bytereef.org/mpdecimal/quickstart.html](http://www.bytereef.org/mpdecimal/quickstart.html).

<table>
<thead>
<tr>
<th></th>
<th>decimal.py</th>
<th>_decimal</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>42.02s</td>
<td>0.345s</td>
<td>120x</td>
</tr>
<tr>
<td>telco</td>
<td>172.19s</td>
<td>5.68s</td>
<td>30x</td>
</tr>
<tr>
<td>psycopg</td>
<td>3.57s</td>
<td>0.29s</td>
<td>12x</td>
</tr>
</tbody>
</table>

Features
• The **FloatOperation** signal optionally enables stricter semantics for mixing floats and Decimals.

• If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the variable **HAVE_THREADS** is set to **False**.

**API changes**

• The C module has the following context limits, depending on the machine architecture:

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_PREC</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>MAX_EMAX</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>MIN_EMIN</td>
<td>-4250000000</td>
<td>-999999999999999999</td>
</tr>
</tbody>
</table>

• In the context templates (**DefaultContext**, **BasicContext** and **ExtendedContext**) the magnitude of Emax and Emin has changed to 999999.

• The **Decimal** constructor in decimal.py does not observe the context limits and converts values with arbitrary exponents or precision exactly. Since the C version has internal limits, the following scheme is used: If possible, values are converted exactly, otherwise **InvalidOperation** is raised and the result is NaN. In the latter case it is always possible to use **create_decimal()** in order to obtain a rounded or inexact value.

• The power function in decimal.py is always correctly-rounded. In the C version, it is defined in terms of the correctly-rounded
exp() and ln() functions, but the final result is only “almost always correctly rounded”.

- In the C version, the context dictionary containing the signals is a MutableMapping. For speed reasons, flags and traps always refer to the same MutableMapping that the context was initialized with. If a new signal dictionary is assigned, flags and traps are updated with the new values, but they do not reference the RHS dictionary.

- Pickling a Context produces a different output in order to have a common interchange format for the Python and C versions.

- The order of arguments in the Context constructor has been changed to match the order displayed by repr().

- The watchexp parameter in the quantize() method is deprecated.

email

Policy Framework

The email package now has a policy framework. A Policy is an object with several methods and properties that control how the email package behaves. The primary policy for Python 3.3 is the Compat32 policy, which provides backward compatibility with the email package in Python 3.2. A policy can be specified when an email message is parsed by a parser, or when a Message object is created, or when an email is serialized using a generator. Unless overridden, a policy passed to a parser is inherited by all the Message object and sub-objects created by the parser. By default a generator will use the
policy of the Message object it is serializing. The default policy is compat32.

The minimum set of controls implemented by all policy objects are:

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_line_length</td>
<td>The maximum length, excluding the linesep character(s), individual lines may have when a Message is serialized. Defaults to 78.</td>
</tr>
<tr>
<td>linesep</td>
<td>The character used to separate individual lines when a Message is serialized. Defaults to \n.</td>
</tr>
<tr>
<td>cte_type</td>
<td>7bit or 8bit. 8bit applies only to a Bytes generator, and means that non-ASCII may be used where allowed by the protocol (or where it exists in the original input).</td>
</tr>
<tr>
<td>raise_on_defect</td>
<td>Causes a parser to raise error when defects are encountered instead of adding them to the Message object's defects list.</td>
</tr>
</tbody>
</table>

A new policy instance, with new settings, is created using the clone() method of policy objects. clone takes any of the above controls as keyword arguments. Any control not specified in the call retains its default value. Thus you can create a policy that uses \r\n linesep characters like this:

```
mypolicy = compat32.clone(linesep='\r\n')
```

Policies can be used to make the generation of messages in the format needed by your application simpler. Instead of having to remember to specify linesep='\r\n' in all the places you call a generator, you can specify it once, when you set the policy used by
the parser or the Message, whichever your program uses to create Message objects. On the other hand, if you need to generate messages in multiple forms, you can still specify the parameters in the appropriate generator call. Or you can have custom policy instances for your different cases, and pass those in when you create the generator.

Provisional Policy with New Header API

While the policy framework is worthwhile all by itself, the main motivation for introducing it is to allow the creation of new policies that implement new features for the email package in a way that maintains backward compatibility for those who do not use the new policies. Because the new policies introduce a new API, we are releasing them in Python 3.3 as a provisional policy. Backwards incompatible changes (up to and including removal of the code) may occur if deemed necessary by the core developers.

The new policies are instances of EmailPolicy, and add the following additional controls:

<table>
<thead>
<tr>
<th>refold_source</th>
<th>Controls whether or not headers parsed by a parser are refolded by the generator. It can be none, long, or all. The default is long, which means that source headers with a line longer than max_line_length get refolded. none means no line get refolded, and all means that all lines get refolded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>header_factory</td>
<td>A callable that take a name and value and produces a custom header object.</td>
</tr>
</tbody>
</table>

The header_factory is the key to the new features provided by the
new policies. When one of the new policies is used, any header retrieved from a `Message` object is an object produced by the `header_factory`, and any time you set a header on a `Message` it becomes an object produced by `header_factory`. All such header objects have a `name` attribute equal to the header name. Address and Date headers have additional attributes that give you access to the parsed data of the header. This means you can now do things like this:

```python
>>> m = Message(policy=SMTP)
>>> m['To'] = 'Éric <foo@example.com>'
>>> m['to']
'Éric <foo@example.com>'
>>> m['to'].addresses
(Address(display_name='Éric', username='foo', domain='example.com'),)
>>> m['to'].addresses[0].username
'foo'
>>> m['to'].addresses[0].display_name
'Éric'
>>> m['Date'] = email.utils.localtime()
>>> m['Date'].datetime
datetime.datetime(2012, 5, 25, 21, 39, 24, 465484, tzinfo=datetime.timezone(datetime.timedelta(-1, 72000), 'EDT'))
>>> m['Date']
'Fri, 25 May 2012 21:44:27 -0400'
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
```

You will note that the unicode display name is automatically encoded as `utf-8` when the message is serialized, but that when the header is accessed directly, you get the unicode version. This eliminates any need to deal with the `email.header decode_header()` or `make_header()` functions.

You can also create addresses from parts:
>>> m['cc'] = [Group('pals', [Address('Bob', 'bob', ... Address('Sally', 'sally', ... Address('Bonzo', addr_spec='bonz@laugh.com'))]

>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
cc: pals: Bob <bob@example.com>, Sally <sally@example.com>,

Decoding to unicode is done automatically:

>>> m2 = message_from_string(str(m))
>>> m2['to']
'Éric <foo@example.com>'

When you parse a message, you can use the addresses and groups attributes of the header objects to access the groups and individual addresses:

>>> m2['cc'].addresses
(Address(display_name='Bob', username='bob', domain='example.com'),
>>> m2['cc'].groups
(Group(display_name='pals', addresses=(Address(display_name='Bob', username='bob', domain='example.com'),

In summary, if you use one of the new policies, header manipulation works the way it ought to: your application works with unicode strings, and the email package transparently encodes and decodes the unicode to and from the RFC standard Content Transfer Encodings.

Other API Changes

New BytesHeaderParser, added to the parser module to complement HeaderParser and complete the Bytes API.
New utility functions:

- **format_datetime()**: given a `datetime`, produce a string formatted for use in an email header.
- **parsedate_to_datetime()**: given a date string from an email header, convert it into an aware `datetime`, or a naive `datetime` if the offset is `-0000`.
- **localtime()**: With no argument, returns the current local time as an aware `datetime` using the local timezone. Given an aware `datetime`, converts it into an aware `datetime` using the local timezone.

**ftplib**

- `ftplib.FTP` now accepts a `source_address` keyword argument to specify the `(host, port)` to use as the source address in the bind call when creating the outgoing socket. (Contributed by Giampaolo Rodolà in bpo-8594.)
- The `FTP_TLS` class now provides a new `ccc()` function to revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports. (Contributed by Giampaolo Rodolà in bpo-12139.)
- Added `ftplib.FTP.mlsd()` method which provides a parsable directory listing format and deprecates `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`. (Contributed by Giampaolo Rodolà in bpo-11072.)

**functools**

The `functools.lru_cache()` decorator now accepts a `typed` keyword argument (that defaults to `False` to ensure that it caches
values of different types that compare equal in separate cache slots. (Contributed by Raymond Hettinger in bpo-13227.)

gc

It is now possible to register callbacks invoked by the garbage collector before and after collection using the new callbacks list.

hmac

A new compare_digest() function has been added to prevent side channel attacks on digests through timing analysis. (Contributed by Nick Coghlan and Christian Heimes in bpo-15061.)

http

http.server.BaseHTTPRequestHandler now buffers the headers and writes them all at once when end_headers() is called. A new method flush_headers() can be used to directly manage when the accumulated headers are sent. (Contributed by Andrew Schaaf in bpo-3709.)

http.server now produces valid HTML 4.01 strict output. (Contributed by Ezio Melotti in bpo-13295.)

http.client.HTTPResponse now has a readinto() method, which means it can be used as an io.RawIOBase class. (Contributed by John Kuhn in bpo-13464.)
html.parser.HTMLParser is now able to parse broken markup without raising errors, therefore the strict argument of the constructor and the HTMLParseError exception are now deprecated. The ability to parse broken markup is the result of a number of bug fixes that are also available on the latest bug fix releases of Python 2.7/3.2. (Contributed by Ezio Melotti in bpo-15114, and bpo-14538, bpo-13993, bpo-13960, bpo-13358, bpo-1745761, bpo-755670, bpo-13357, bpo-12629, bpo-1200313, bpo-670664, bpo-13273, bpo-12888, bpo-7311.)

A new html5 dictionary that maps HTML5 named character references to the equivalent Unicode character(s) (e.g. html5['gt;'] == '>') has been added to the html.entities module. The dictionary is now also used by HTMLParser. (Contributed by Ezio Melotti in bpo-11113 and bpo-15156.)

imaplib

The IMAP4_SSL constructor now accepts an SSLContext parameter to control parameters of the secure channel.

(Contributed by Sijin Joseph in bpo-8808.)

inspect

A new getclosurevars() function has been added. This function reports the current binding of all names referenced from the function body and where those names were resolved, making it easier to verify correct internal state when testing code that relies on stateful closures.

(Contributed by Meador Inge and Nick Coghlan in bpo-13062.)

A new getgeneratorlocals() function has been added. This
function reports the current binding of local variables in the generator’s stack frame, making it easier to verify correct internal state when testing generators.

(Contributed by Meador Inge in bpo-15153.)

**io**

The `open()` function has a new 'x' mode that can be used to exclusively create a new file, and raise a `FileExistsError` if the file already exists. It is based on the C11 ‘x’ mode to fopen().

(Contributed by David Townshend in bpo-12760.)

The constructor of the `TextIOWrapper` class has a new `write_through` optional argument. If `write_through` is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the `TextIOWrapper` object is immediately handled to its underlying binary buffer.

**itertools**

`accumulate()` now takes an optional `func` argument for providing a user-supplied binary function.

**logging**

The `basicConfig()` function now supports an optional `handlers` argument taking an iterable of handlers to be added to the root logger.

A class level attribute `append_nul` has been added to
SysLogHandler to allow control of the appending of the NUL (\000) byte to syslog records, since for some daemons it is required while for others it is passed through to the log.

**math**

The *math* module has a new function, *log2()*, which returns the base-2 logarithm of *x*.

(Written by Mark Dickinson in bpo-11888.)

**mmap**

The *read()* method is now more compatible with other file-like objects: if the argument is omitted or specified as *None*, it returns the bytes from the current file position to the end of the mapping. (Contributed by Petri Lehtinen in bpo-12021.)

**multiprocessing**

The new *multiprocessing.connection.wait()* function allows polling multiple objects (such as connections, sockets and pipes) with a timeout. (Contributed by Richard Oudkerk in bpo-12328.)

*multiprocessing.Connection* objects can now be transferred over *multiprocessing* connections. (Contributed by Richard Oudkerk in bpo-4892.)

*multiprocessing.Process* now accepts a *daemon* keyword argument to override the default behavior of inheriting the daemon flag from the parent process (bpo-6064).
New attribute `multiprocessing.Process.sentinel` allows a program to wait on multiple `Process` objects at one time using the appropriate OS primitives (for example, `select` on posix systems).

New methods `multiprocessing.pool.Pool.starmap()` and `starmap_async()` provide `itertools.starmap()` equivalents to the existing `multiprocessing.pool.Pool.map()` and `map_async()` functions. (Contributed by Hynek Schlawack in `bpo-12708`.)

**nntplib**

The `nntplib.NNTP` class now supports the context management protocol to unconditionally consume `socket.error` exceptions and to close the NNTP connection when done:

```python
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
... ('211 1755 1 1755 gmane.comp.python.committers',)
```

(Contributed by Giampaolo Rodolà in `bpo-9795`.)

**OS**

- The `os` module has a new `pipe2()` function that makes it possible to create a pipe with `O_CLOEXEC` or `O_NONBLOCK` flags set atomically. This is especially useful to avoid race conditions in multi-threaded programs.
The `os` module has a new `sendfile()` function which provides an efficient “zero-copy” way for copying data from one file (or socket) descriptor to another. The phrase “zero-copy” refers to the fact that all of the copying of data between the two descriptors is done entirely by the kernel, with no copying of data into userspace buffers. `sendfile()` can be used to efficiently copy data from a file on disk to a network socket, e.g. for downloading a file.

(Patch submitted by Ross Lagerwall and Giampaolo Rodolà in `bpo-10882`.)

To avoid race conditions like symlink attacks and issues with temporary files and directories, it is more reliable (and also faster) to manipulate file descriptors instead of file names. Python 3.3 enhances existing functions and introduces new functions to work on file descriptors (`bpo-4761`, `bpo-10755` and `bpo-14626`).

- The `os` module has a new `fwalk()` function similar to `walk()` except that it also yields file descriptors referring to the directories visited. This is especially useful to avoid symlink races.
- The following functions get new optional `dir_fd` (paths relative to directory descriptors) and/or `follow_symlinks` (not following symlinks): `access()`, `chflags()`, `chmod()`, `chown()`, `link()`, `lstat()`, `mkdir()`, `mkfifo()`, `mknod()`, `open()`, `readlink()`, `remove()`, `rename()`, `replace()`, `rmdir()`, `stat()`, `symlink()`, `unlink()`, `utime()`. Platform support for using these parameters can be checked via the sets `os.supports_dir_fd` and `os.supports_follows_symlinks`.
- The following functions now support a file descriptor for their path argument: `chdir()`, `chmod()`, `chown()`, `execve()`, `listdir()`, `pathconf()`, `exists()`, `stat()`, `fstat()`.
**statvfs()**, **utime()**. Platform support for this can be checked via the `os.supports_fd` set.

- **access()** accepts an `effective_ids` keyword argument to turn on using the effective uid/gid rather than the real uid/gid in the access check. Platform support for this can be checked via the `supports_effective_ids` set.

- The `os` module has two new functions: `getpriority()` and `setpriority()`. They can be used to get or set process niceness/priority in a fashion similar to `os.nice()` but extended to all processes instead of just the current one.

(Patch submitted by Giampaolo Rodolà in bpo-10784.)

- The new `os.replace()` function allows cross-platform renameing of a file with overwriting the destination. With `os.rename()`, an existing destination file is overwritten under POSIX, but raises an error under Windows. (Contributed by Antoine Pitrou in bpo-8828.)

- The stat family of functions (**stat()**, **fstat()**, and **lstat()**) now support reading a file's timestamps with nanosecond precision. Symmetrically, **utime()** can now write file timestamps with nanosecond precision. (Contributed by Larry Hastings in bpo-14127.)

- The new `os.get_terminal_size()` function queries the size of the terminal attached to a file descriptor. See also `shutil.get_terminal_size()`.(Contributed by Zbigniew Jędrzejewski-Szmek in bpo-13609.)

- New functions to support Linux extended attributes (bpo-12720): `getxattr()`, `listxattr()`, `removexattr()`, `setxattr()`.
• New interface to the scheduler. These functions control how a process is allocated CPU time by the operating system. New functions:
  - `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getaffinity()`, `sched_getparam()`, `sched_getscheduler()`, `sched_rr_get_interval()`, `sched_setaffinity()`, `sched_setparam()`, `sched_setscheduler()`, `sched_yield()`,
• New functions to control the file system:
  - `posix_fadvise()`: Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations.
  - `posix_fallocate()`: Ensures that enough disk space is allocated for a file.
  - `sync()`: Force write of everything to disk.
• Additional new posix functions:
  - `lockf()`: Apply, test or remove a POSIX lock on an open file descriptor.
  - `pread()`: Read from a file descriptor at an offset, the file offset remains unchanged.
  - `pwrite()`: Write to a file descriptor from an offset, leaving the file offset unchanged.
  - `readv()`: Read from a file descriptor into a number of writable buffers.
  - `truncate()`: Truncate the file corresponding to `path`, so that it is at most `length` bytes in size.
  - `waitid()`: Wait for the completion of one or more child processes.
  - `writev()`: Write the contents of `buffers` to a file descriptor, where `buffers` is an arbitrary sequence of buffers.
  - `getgroupolist()` (bpo-9344): Return list of group ids that specified user belongs to.
• `times()` and `uname()`: Return type changed from a tuple to a
tuple-like object with named attributes.

- Some platforms now support additional constants for the `lseek()` function, such as `os.SEEK_HOLE` and `os.SEEK_DATA`.
- New constants `RTLD_LAZY`, `RTLD_NOW`, `RTLD_GLOBAL`, `RTLD_LOCAL`, `RTLD_NODELETE`, `RTLD_NOLOAD`, and `RTLD_DEEPBIND` are available on platforms that support them. These are for use with the `sys.setdlopenflags()` function, and supersede the similar constants defined in `ctypes` and `DLFCN`. (Contributed by Victor Stinner in bpo-13226.)
- `os.symlink()` now accepts (and ignores) the `target_is_directory` keyword argument on non-Windows platforms, to ease cross-platform support.

**pdb**

Tab-completion is now available not only for command names, but also their arguments. For example, for the `break` command, function and file names are completed.

(Contributed by Georg Brandl in bpo-14210)

**pickle**

`pickle.Pickler` objects now have an optional `dispatch_table` attribute allowing per-pickler reduction functions to be set.

(Contributed by Richard Oudkerk in bpo-14166.)

**pydoc**

The Tk GUI and the `serve()` function have been removed from the
**pydoc** module: `pydoc -g` and `serve()` have been deprecated in Python 3.2.

**re**

`str` regular expressions now support \u and \U escapes.

(Contributed by Serhiy Storchaka in bpo-3665.)

**sched**

- `run()` now accepts a *blocking* parameter which when set to false makes the method execute the scheduled events due to expire soonest (if any) and then return immediately. This is useful in case you want to use the `scheduler` in non-blocking applications. (Contributed by Giampaolo Rodolà in bpo-13449.)
- `scheduler` class can now be safely used in multi-threaded environments. (Contributed by Josiah Carlson and Giampaolo Rodolà in bpo-8684.)
- `timefunc` and `delayfunc` parameters of `scheduler` class constructor are now optional and defaults to `time.time()` and `time.sleep()` respectively. (Contributed by Chris Clark in bpo-13245.)
- `enter()` and `enterabs()` *argument* parameter is now optional. (Contributed by Chris Clark in bpo-13245.)
- `enter()` and `enterabs()` now accept a *kwargs* parameter. (Contributed by Chris Clark in bpo-13245.)

**select**

Solaris and derivative platforms have a new class `select.devpoll` for high performance asynchronous sockets via `/dev/poll`. 
The previously undocumented helper function `quote` from the `pipes` modules has been moved to the `shlex` module and documented. `quote()` properly escapes all characters in a string that might be otherwise given special meaning by the shell.

**shutil**

- New functions:
  - `disk_usage()`: provides total, used and free disk space statistics. (Contributed by Giampaolo Rodolà in bpo-12442.)
  - `chown()`: allows one to change user and/or group of the given path also specifying the user/group names and not only their numeric ids. (Contributed by Sandro Tosi in bpo-12191.)
  - `shutil.get_terminal_size()`: returns the size of the terminal window to which the interpreter is attached. (Contributed by Zbigniew Jędrzejewski-Szmek in bpo-13609.)
- `copy2()` and `copystat()` now preserve file timestamps with nanosecond precision on platforms that support it. They also preserve file “extended attributes” on Linux. (Contributed by Larry Hastings in bpo-14127 and bpo-15238.)
- Several functions now take an optional `symlinks` argument: when that parameter is true, symlinks aren’t dereferenced and the operation instead acts on the symlink itself (or creates one, if relevant). (Contributed by Hynek Schlawack in bpo-12715.)
- When copying files to a different file system, `move()` now handles symlinks the way the posix `mv` command does, recreating the symlink rather than copying the target file contents. (Contributed by Jonathan Niehof in bpo-9993.) `move()` now also
returns the dst argument as its result.

- `rmtree()` is now resistant to symlink attacks on platforms which support the new `dir_fd` parameter in `os.open()` and `os.unlink()`. (Contributed by Martin von Löwis and Hynek Schlawack in bpo-4489.)

**signal**

- The `signal` module has new functions:
  - `pthread_sigmask()`: fetch and/or change the signal mask of the calling thread (Contributed by Jean-Paul Calderone in bpo-8407);
  - `pthread_kill()`: send a signal to a thread;
  - `sigpending()`: examine pending functions;
  - `sigwait()`: wait a signal;
  - `sigwaitinfo()`: wait for a signal, returning detailed information about it;
  - `sigtimedwait()`: like `sigwaitinfo()` but with a timeout.

- The signal handler writes the signal number as a single byte instead of a nul byte into the wakeup file descriptor. So it is possible to wait more than one signal and know which signals were raised.

- `signal.signal()` and `signal.siginterrupt()` raise an OSError, instead of a RuntimeError: OSError has an errno attribute.

**smtpd**

The `smtpd` module now supports RFC 5321 (extended SMTP) and RFC 1870 (size extension). Per the standard, these extensions are enabled if and only if the client initiates the session with an EHLO command.
(Initial ELHO support by Alberto Trevino. Size extension by Juhana Jauhiainen. Substantial additional work on the patch contributed by Michele Orrù and Dan Boswell. bpo-8739)

**smtplib**

The **SMTP**, **SMTP_SSL**, and **LMTP** classes now accept a `source_address` keyword argument to specify the (host, port) to use as the source address in the bind call when creating the outgoing socket. (Contributed by Paulo Scardine in bpo-11281.)

**SMTP** now supports the context management protocol, allowing an SMTP instance to be used in a `with` statement. (Contributed by Giampaolo Rodolà in bpo-11289.)

The **SMTP_SSL** constructor and the `starttls()` method now accept an SSLContext parameter to control parameters of the secure channel. (Contributed by Kasun Herath in bpo-8809.)

**socket**

- The **socket** class now exposes additional methods to process ancillary data when supported by the underlying platform:
  - `sendmsg()`
  - `recvmsg()`
  - `recvmsg_into()`

  (Contributed by David Watson in bpo-6560, based on an earlier patch by Heiko Wundram)

- The **socket** class now supports the PF_CAN protocol family

The `socket` class now supports the PF_SYSTEM protocol family on OS X. (Contributed by Michael Goderbauer in bpo-13777.)

New function `sethostname()` allows the hostname to be set on unix systems if the calling process has sufficient privileges. (Contributed by Ross Lagerwall in bpo-10866.)

**socketserver**

`BaseServer` now has an overridable method `service_actions()` that is called by the `serve_forever()` method in the service loop. `ForkingMixIn` now uses this to clean up zombie child processes. (Contributed by Justin Warkentin in bpo-11109.)

**sqlite3**

New `sqlite3.Connection` method `set_trace_callback()` can be used to capture a trace of all sql commands processed by sqlite. (Contributed by Torsten Landschoff in bpo-11688.)

**ssl**
The `ssl` module has two new random generation functions:

- `RAND_bytes()`: generate cryptographically strong pseudo-random bytes.
- `RAND_pseudo_bytes()`: generate pseudo-random bytes.

(Contributed by Victor Stinner in bpo-12049.)

The `ssl` module now exposes a finer-grained exception hierarchy in order to make it easier to inspect the various kinds of errors. (Contributed by Antoine Pitrou in bpo-11183.)

`load_cert_chain()` now accepts a `password` argument to be used if the private key is encrypted. (Contributed by Adam Simpkins in bpo-12803.)

Diffie-Hellman key exchange, both regular and Elliptic Curve-based, is now supported through the `load_dh_params()` and `set_ecdh_curve()` methods. (Contributed by Antoine Pitrou in bpo-13626 and bpo-13627.)

SSL sockets have a new `get_channel_binding()` method allowing the implementation of certain authentication mechanisms such as SCRAM-SHA-1-PLUS. (Contributed by Jacek Konieczny in bpo-12551.)

You can query the SSL compression algorithm used by an SSL socket, thanks to its new `compression()` method. The new attribute `OP_NO_COMPRESSION` can be used to disable compression. (Contributed by Antoine Pitrou in bpo-13634.)

Support has been added for the Next Protocol Negotiation extension using the
ssl.SSLContext.set_npn_protocols() method.
(Contributed by Colin Marc in bpo-14204.)

- SSL errors can now be introspected more easily thanks to library and reason attributes. (Contributed by Antoine Pitrou in bpo-14837.)

- The get_server_certificate() function now supports IPv6. (Contributed by Charles-François Natali in bpo-11811.)

- New attribute OP_CIPHER_SERVER_PREFERENCE allows setting SSLv3 server sockets to use the server’s cipher ordering preference rather than the client's (bpo-13635).

stat

The undocumented tarfile.filemode function has been moved to stat.filemode(). It can be used to convert a file’s mode to a string of the form '-rwxrwxrwx'.

(Contributed by Giampaolo Rodolà in bpo-14807.)

struct

The struct module now supports ssize_t and size_t via the new codes n and N, respectively. (Contributed by Antoine Pitrou in bpo-3163.)

subprocess

Command strings can now be bytes objects on posix platforms. (Contributed by Victor Stinner in bpo-8513.)
A new constant `DEVNULL` allows suppressing output in a platform-independent fashion. (Contributed by Ross Lagerwall in bpo-5870.)

sys

The `sys` module has a new `thread_info struct sequence` holding information about the thread implementation (bpo-11223).

tarfile

`tarfile` now supports `lzma` encoding via the `lzma` module. (Contributed by Lars Gustäbel in bpo-5689.)

tempfile

`tempfile.SpooledTemporaryFile`'s `truncate()` method now accepts a `size` parameter. (Contributed by Ryan Kelly in bpo-9957.)

textwrap

The `textwrap` module has a new `indent()` that makes it straightforward to add a common prefix to selected lines in a block of text (bpo-13857).

threading

`threading.Condition`, `threading.Semaphore`, `threading.BoundedSemaphore`, `threading.Event`, and `threading.Timer`, all of which used to be factory functions returning a class instance, are now classes and may be subclassed.
The `threading.Thread` constructor now accepts a daemon keyword argument to override the default behavior of inheriting the daemon flag value from the parent thread (bpo-6064).

The formerly private function `_thread.get_ident` is now available as the public function `threading.get_ident()`. This eliminates several cases of direct access to the `_thread` module in the stdlib. Third party code that used `_thread.get_ident` should likewise be changed to use the new public interface.

### time

The [PEP 418](https://www.python.org/dev/peps/pep-0418/) added new functions to the `time` module:

- `get_clock_info()`: Get information on a clock.
- `monotonic()`: Monotonic clock (cannot go backward), not affected by system clock updates.
- `perf_counter()`: Performance counter with the highest available resolution to measure a short duration.
- `process_time()`: Sum of the system and user CPU time of the current process.

Other new functions:

- `clock_getres()`, `clock_gettime()` and `clock_settime()` functions with `CLOCK_xxx` constants. (Contributed by Victor Stinner in bpo-10278.)

To improve cross platform consistency, `sleep()` now raises a `ValueError` when passed a negative sleep value. Previously this was an error on posix, but produced an infinite sleep on Windows.
types

Add a new `types.MappingProxyType` class: Read-only proxy of a mapping. (bpo-14386)

The new functions `types.new_class()` and `types.prepare_class()` provide support for PEP 3115 compliant dynamic type creation. (bpo-14588)

unittest

`assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, and `assertWarnsRegex()` now accept a keyword argument `msg` when used as context managers. (Contributed by Ezio Melotti and Winston Ewert in bpo-10775.)

`unittest.TestCase.run()` now returns the `TestResult` object.

urllib

The `Request` class, now accepts a `method` argument used by `get_method()` to determine what HTTP method should be used. For example, this will send a 'HEAD' request:

```python
>>> urlopen(Request('https://www.python.org', method='HEAD'))
```

(bpo-1673007)

webbrowser
The `webbrowser` module supports more “browsers”: Google Chrome (named `chrome`, `chromium`, `chrome-browser` or `chromium-browser` depending on the version and operating system), and the generic launchers `xdg-open`, from the FreeDesktop.org project, and `gvfs-open`, which is the default URI handler for GNOME 3. (The former contributed by Arnaud Calmettes in bpo-13620, the latter by Matthias Klose in bpo-14493.)

**xml.etree.ElementTree**

The `xml.etree.ElementTree` module now imports its C accelerator by default; there is no longer a need to explicitly import `xml.etree.cElementTree` (this module stays for backwards compatibility, but is now deprecated). In addition, the `iter` family of methods of `Element` has been optimized (rewritten in C). The module’s documentation has also been greatly improved with added examples and a more detailed reference.

**zlib**

New attribute `zlib.Decompress.eof` makes it possible to distinguish between a properly-formed compressed stream and an incomplete or truncated one. (Contributed by Nadeem Vawda in bpo-12646.)

New attribute `zlib.ZLIB_RUNTIME_VERSION` reports the version string of the underlying `zlib` library that is loaded at runtime. (Contributed by Torsten Landschoff in bpo-12306.)
Optimizations

Major performance enhancements have been added:

- Thanks to PEP 393, some operations on Unicode strings have been optimized:
  - the memory footprint is divided by 2 to 4 depending on the text
  - encode an ASCII string to UTF-8 doesn't need to encode characters anymore, the UTF-8 representation is shared with the ASCII representation
  - the UTF-8 encoder has been optimized
  - repeating a single ASCII letter and getting a substring of an ASCII string is 4 times faster
- UTF-8 is now 2x to 4x faster. UTF-16 encoding is now up to 10x faster.

(Contributed by Serhiy Storchaka, bpo-14624, bpo-14738 and bpo-15026.)
Build and C API Changes

Changes to Python's build process and to the C API include:

- New **PEP 3118** related function:
  - `PyMemoryView_FromMemory()`
- **PEP 393** added new Unicode types, macros and functions:
  - **High-level API:**
    - `PyUnicode_CopyCharacters()`
    - `PyUnicode_FindChar()`
    - `PyUnicode_GetLength()`, `PyUnicode_GET_LENGTH`
    - `PyUnicode_New()`
    - `PyUnicode_Substring()`
    - `PyUnicode_ReadChar()`, `PyUnicode_WriteChar()`
  - **Low-level API:**
    - `Py_UCS1`, `Py_UCS2`, `Py_UCS4` types
    - `PyASCIIObject` and `PyCompactUnicodeObject` structures
    - `PyUnicode_READY`
    - `PyUnicode_FromKindAndData()`
    - `PyUnicode_AsUCS4()`, `PyUnicode_AsUCS4Copy()`
    - `PyUnicode_DATA`, `PyUnicode_1BYTE_DATA`, `PyUnicode_2BYTE_DATA`, `PyUnicode_4BYTE_DATA`
    - `PyUnicode_KIND` with `PyUnicode_KIND` enum:
      - `PyUnicode_WCHAR_KIND`, `PyUnicode_1BYTE_KIND`, `PyUnicode_2BYTE_KIND`, `PyUnicode_4BYTE_KIND`
    - `PyUnicode_READ`, `PyUnicode_READ_CHAR`, `PyUnicode_WRITE`
    - `PyUnicode_MAX_CHAR_VALUE`
• `PyArg_ParseTuple` now accepts a `bytearray` for the c format (bpo-12380).
Deprecated

Unsupported Operating Systems

OS/2 and VMS are no longer supported due to the lack of a maintainer.

Windows 2000 and Windows platforms which set `COMSPEC` to `command.com` are no longer supported due to maintenance burden.

OSF support, which was deprecated in 3.2, has been completely removed.

Deprecated Python modules, functions and methods

- Passing a non-empty string to `object.__format__()` is deprecated, and will produce a `TypeError` in Python 3.4 (bpo-9856).
- The `unicode_internal` codec has been deprecated because of the PEP 393, use UTF-8, UTF-16 (utf-16-le or utf-16-be), or UTF-32 (utf-32-le or utf-32-be)
- `ftplib.FTP.nlst()` and `ftplib.FTP.dir()` use `ftplib.FTP.mlsd()`.
- `platform.popen()`: use the `subprocess` module. Check especially the Replacing Older Functions with the subprocess Module section (bpo-11377).
- bpo-13374: The Windows bytes API has been deprecated in the `os` module. Use Unicode filenames, instead of bytes filenames, to not depend on the ANSI code page anymore and to support any filename.
- bpo-13988: The `xml.etree.cElementTree` module is
deprecated. The accelerator is used automatically whenever available.

- The behaviour of `time.clock()` depends on the platform: use the new `time.perf_counter()` or `time.process_time()` function instead, depending on your requirements, to have a well defined behaviour.
- The `os.stat_float_times()` function is deprecated.
- `abc` module:
  - `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
  - `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
  - `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.
- `importlib` package:
  - `importlib.abc.SourceLoader.path_mtime()` is now deprecated in favour of `importlib.abc.SourceLoader.path_stats()` as bytecode files now store both the modification time and size of the source file the bytecode file was compiled from.

## Deprecated functions and types of the C API

The `Py_UNICODE` has been deprecated by [PEP 393](https://www.python.org/dev/peps/pep-0393/) and will be removed in Python 4. All functions using this type are deprecated:

### Unicode functions and methods using `Py_UNICODE` and `Py_UNICODE*` types:

- `PyUnicode_FromUnicode`: use `PyUnicode_FromWideChar()` or `PyUnicode_FromKindAndData()`
PyUnicode_AS_UNICODE, PyUnicode_AsUnicode(),
PyUnicode_AsUnicodeAndSize(): use
PyUnicode_AsWideCharString()
PyUnicode_AS_DATA: use PyUnicode_DATA with
PyUnicode_READ and PyUnicode_WRITE
PyUnicode_GET_SIZE, PyUnicode_GetSize(): use
PyUnicode_GET_LENGTH or PyUnicode_GetLength()
PyUnicode_GET_DATA_SIZE: use
PyUnicode_GET_LENGTH(str) \* PyUnicode_KIND(str)
(only work on ready strings)
PyUnicode_AsUnicodeCopy(): use
PyUnicode_AsUCS4Copy() or
PyUnicode_AsWideCharString()
PyUnicode_GetMax()

Functions and macros manipulating Py_UNICODE* strings:

Py_UNICODE_strlen: use PyUnicode.GetLength() or
PyUnicode_GET_LENGTH
Py_UNICODE_strcat: use PyUnicode_CopyCharacters() or
PyUnicode_FromFormat()
Py_UNICODE_strcpy, Py_UNICODE_strncpy, Py_UNICODE_COPY: use
PyUnicode_CopyCharacters() or
PyUnicode_Substring()
Py_UNICODE_strcmp: use PyUnicode_Compare()
Py_UNICODE_strncmp: use PyUnicode_Tailmatch()
Py_UNICODE_strchr, Py_UNICODE_strrchr: use
PyUnicode_FindChar()
Py_UNICODE_FILL: use PyUnicode.Fill()
Py_UNICODE_MATCH

Encoders:
- `PyUnicode_Encode()`: use `PyUnicode_AsEncodedObject()``
- `PyUnicode_EncodeUTF7()`
- `PyUnicode_EncodeUTF8()`: use `PyUnicode_AsUTF8()` or `PyUnicode_AsUTF8String()`
- `PyUnicode_EncodeUTF32()`
- `PyUnicode_EncodeUTF16()`
- `PyUnicode_EncodeUnicodeEscape():` use `PyUnicode_AsUnicodeEscapeString()`
- `PyUnicode_EncodeRawUnicodeEscape():` use `PyUnicode_AsRawUnicodeEscapeString()`
- `PyUnicode_EncodeLatin1():` use `PyUnicode_AsLatin1String()`
- `PyUnicode_EncodeASCII():` use `PyUnicode_AsASCIIString()`
- `PyUnicode_EncodeCharmap()`
- `PyUnicode_TranslateCharmap()`
- `PyUnicode_EncodeMBCS():` use `PyUnicode_AsMBCSString()` or `PyUnicode_EncodeCodePage()` (with `CP_ACP` code_page)
- `PyUnicode_EncodeDecimal()`, `PyUnicode_TransformDecimalToASCII()`

**Deprecated features**

The `array` module's `u` format code is now deprecated and will be removed in Python 4 together with the rest of the `(Py_UNICODE)` API.
Porting to Python 3.3

This section lists previously described changes and other bug fixes that may require changes to your code.

Porting Python code

- Hash randomization is enabled by default. Set the `PYTHONHASHSEED` environment variable to 0 to disable hash randomization. See also the `object.__hash__()` method.
- **bpo-12326**: On Linux, `sys.platform` doesn’t contain the major version anymore. It is now always ‘linux’, instead of ‘linux2’ or ‘linux3’ depending on the Linux version used to build Python. Replace `sys.platform == ‘linux2’` with `sys.platform.startswith(‘linux’), or directly `sys.platform == ‘linux’` if you don’t need to support older Python versions.
- **bpo-13847, bpo-14180**: `time` and `datetime`: `OverflowError` is now raised instead of `ValueError` if a timestamp is out of range. `OSError` is now raised if C functions `gmtime()` or `localtime()` failed.
- The default finders used by `import` now utilize a cache of what is contained within a specific directory. If you create a Python source file or sourceless bytecode file, make sure to call `importlib.invalidate_caches()` to clear out the cache for the finders to notice the new file.
- `ImportError` now uses the full name of the module that was attempted to be imported. Doctests that check `ImportErrors’ message will need to be updated to use the full name of the module instead of just the tail of the name.
- The `index` argument to `__import__() `now defaults to 0 instead of -1 and no longer support negative values. It was an oversight when `PEP 328` was implemented that the default value remained
-1. If you need to continue to perform a relative import followed by an absolute import, then perform the relative import using an index of 1, followed by another import using an index of 0. It is preferred, though, that you use `importlib.import_module()` rather than call `__import__()` directly.

- `__import__() no longer allows one to use an index value other than 0 for top-level modules. E.g. `__import__('sys', level=1)` is now an error.

- Because `sys.meta_path` and `sys.path_hooks` now have finders on them by default, you will most likely want to use `list.insert()` instead of `list.append()` to add to those lists.

- Because `None` is now inserted into `sys.path_importer_cache`, if you are clearing out entries in the dictionary of paths that do not have a finder, you will need to remove keys paired with values of `None` and `imp.NullImporter` to be backwards-compatible. This will lead to extra overhead on older versions of Python that re-insert `None` into `sys.path_importer_cache` where it represents the use of implicit finders, but semantically it should not change anything.

- `importlib.abc.Finder` no longer specifies a `find_module()` abstract method that must be implemented. If you were relying on subclasses to implement that method, make sure to check for the method’s existence first. You will probably want to check for `find_loader()` first, though, in the case of working with path entry finders.

- `pkgutil` has been converted to use `importlib` internally. This eliminates many edge cases where the old behaviour of the PEP 302 import emulation failed to match the behaviour of the real import system. The import emulation itself is still present, but is now deprecated. The `pkgutil.iter_importers()` and `pkgutil.walk_packages()` functions special case the standard import hooks so they are still supported even though
they do not provide the non-standard `iter_modules()` method.

- A longstanding RFC-compliance bug (bpo-1079) in the parsing done by `email.header.decode_header()` has been fixed. Code that uses the standard idiom to convert encoded headers into unicode (`str(make_header(decode_header(h)))`) will see no change, but code that looks at the individual tuples returned by `decode_header` will see that whitespace that precedes or follows ASCII sections is now included in the ASCII section. Code that builds headers using `make_header` should also continue to work without change, since `make_header` continues to add whitespace between ASCII and non-ASCII sections if it is not already present in the input strings.

- `email.utils.formataddr()` now does the correct content transfer encoding when passed non-ASCII display names. Any code that depended on the previous buggy behavior that preserved the non-ASCII unicode in the formatted output string will need to be changed (bpo-1690608).

- `poplib.POP3.quit()` may now raise protocol errors like all other `poplib` methods. Code that assumes `quit` does not raise `poplib.error_proto` errors may need to be changed if errors on `quit` are encountered by a particular application (bpo-11291).

- The `strict` argument to `email.parser.Parser`, deprecated since Python 2.4, has finally been removed.

- The deprecated method `unittest.TestCase.assertSameElements` has been removed.

- The deprecated variable `time.accept2dyear` has been removed.

- The deprecated `Context._clamp` attribute has been removed from the `decimal` module. It was previously replaced by the public attribute `clamp`. (See bpo-8540.)

- The undocumented internal helper class `SSLFakeFile` has been
removed from smtplib, since its functionality has long been provided directly by socket.socket.makefile().

- Passing a negative value to time.sleep() on Windows now raises an error instead of sleeping forever. It has always raised an error on posix.
- The ast.__version__ constant has been removed. If you need to make decisions affected by the AST version, use sys.version_info to make the decision.
- Code that used to work around the fact that the threading module used factory functions by subclassing the private classes will need to change to subclass the now-public classes.
- The undocumented debugging machinery in the threading module has been removed, simplifying the code. This should have no effect on production code, but is mentioned here in case any application debug frameworks were interacting with it (bpo-13550).

Porting C code

- In the course of changes to the buffer API the undocumented smalltable member of the Py_buffer structure has been removed and the layout of the PyMemoryViewObject has changed.

  All extensions relying on the relevant parts in memoryobject.h or object.h must be rebuilt.

- Due to PEP 393, the Py_UNICODE type and all functions using this type are deprecated (but will stay available for at least five years). If you were using low-level Unicode APIs to construct and access unicode objects and you want to benefit of the memory footprint reduction provided by PEP 393, you have to convert your
code to the new Unicode API.

However, if you only have been using high-level functions such as `PyUnicode_Concat()`, `PyUnicode_Join()` or `PyUnicode_FromFormat()`, your code will automatically take advantage of the new unicode representations.

- `PyImport_GetMagicNumber()` now returns \(-1\) upon failure.

- As a negative value for the `level` argument to `__import__()` is no longer valid, the same now holds for `PyImport_ImportModuleLevel()`. This also means that the value of `level` used by `PyImport_ImportModuleEx()` is now 0 instead of \(-1\).

**Building C extensions**

- The range of possible file names for C extensions has been narrowed. Very rarely used spellings have been suppressed: under POSIX, files named `xxxmodule.so`, `xxxmodule.abi3.so` and `xxxmodule.cpython-* .so` are no longer recognized as implementing the `xxx` module. If you had been generating such files, you have to switch to the other spellings (i.e., remove the `module` string from the file names).

  (implemented in bpo-14040.)

**Command Line Switch Changes**

- The `-Q` command-line flag and related artifacts have been removed. Code checking `sys.flags.division_warning` will need updating.
When `python` is started with `-S`, `import site` will no longer add site-specific paths to the module search paths. In previous versions, it did.

(bpo-10998, contributed by Éric Araujo.)

(bpo-11591, contributed by Carl Meyer with editions by Éric Araujo.)
What’s New In Python 3.2

Author: Raymond Hettinger

This article explains the new features in Python 3.2 as compared to 3.1. It focuses on a few highlights and gives a few examples. For full details, see the Misc/NEWS file.

See also: PEP 392 - Python 3.2 Release Schedule
PEP 384: Defining a Stable ABI

In the past, extension modules built for one Python version were often not usable with other Python versions. Particularly on Windows, every feature release of Python required rebuilding all extension modules that one wanted to use. This requirement was the result of the free access to Python interpreter internals that extension modules could use.

With Python 3.2, an alternative approach becomes available: extension modules which restrict themselves to a limited API (by defining Py_LIMITED_API) cannot use many of the internals, but are constrained to a set of API functions that are promised to be stable for several releases. As a consequence, extension modules built for 3.2 in that mode will also work with 3.3, 3.4, and so on. Extension modules that make use of details of memory structures can still be built, but will need to be recompiled for every feature release.

See also:

PEP 384 - Defining a Stable ABI
PEP written by Martin von Löwis.
PEP 389: Argparse Command Line Parsing Module

A new module for command line parsing, `argparse`, was introduced to overcome the limitations of `optparse` which did not provide support for positional arguments (not just options), subcommands, required options and other common patterns of specifying and validating options.

This module has already had widespread success in the community as a third-party module. Being more fully featured than its predecessor, the `argparse` module is now the preferred module for command-line processing. The older module is still being kept available because of the substantial amount of legacy code that depends on it.

Here’s an annotated example parser showing features like limiting results to a set of choices, specifying a `metavar` in the help screen, validating that one or more positional arguments is present, and making a required option:

```python
import argparse
parser = argparse.ArgumentParser(  
description = 'Manage servers',  
epilog = 'Tested on Solaris and Linux')
parser.add_argument('action',  
choices = ['deploy', 'start', 'stop'],  
help = 'action on each target')
parser.add_argument('targets',  
metavar = 'HOSTNAME',  
nargs = '+',  
help = 'url for target machines')
parser.add_argument('-u', '--user',  
required = True,)```
help = 'login as user')

Example of calling the parser on a command string:

```python
>>> cmd = 'deploy sneezy.example.com sleepy.example.com -u skycaptain
>>> result = parser.parse_args(cmd.split())
>>> result.action
'deploy'
>>> result.targets
['sneezy.example.com', 'sleepy.example.com']
>>> result.user
'skycaptain'
```

Example of the parser's automatically generated help:

```python
>>> parser.parse_args('-h'.split())
usage: manage_cloud.py [-h] -u USER
    {deploy,start,stop} HOSTNAME

Manage servers

positional arguments:
  {deploy,start,stop} action on each target
  HOSTNAME url for target machines

optional arguments:
  -h, --help                  show this help message and exit
  -u USER, --user USER        login as user

Tested on Solaris and Linux
```

An especially nice `argparse` feature is the ability to define subparsers, each with their own argument patterns and help displays:
import argparse
parser = argparse.ArgumentParser(prog='HELM')
subparsers = parser.add_subparsers()

parser_l = subparsers.add_parser('launch', help='Launch Control')
parser_l.add_argument('-m', '--missiles', action='store_true')
parser_l.add_argument('-t', '--torpedos', action='store_true')

parser_m = subparsers.add_parser('move', help='Move Vessel')
parser_m.add_argument('-c', '--course', type=int, required=True)
parser_m.add_argument('-s', '--speed', type=int, default=1)

$ ./helm.py --help  # top level help
$ ./helm.py launch --help  # help for launch options
$ ./helm.py launch --missiles  # set missiles=True and torpedos=False
$ ./helm.py steer --course 180 --speed 5  # set movement parameters

See also:

**PEP 389 - New Command Line Parsing Module**

PEP written by Steven Bethard.

Upgrading **optparse code** for details on the differences from optparse.
PEP 391: Dictionary Based Configuration for Logging

The `logging` module provided two kinds of configuration, one style with function calls for each option or another style driven by an external file saved in a `ConfigParser` format. Those options did not provide the flexibility to create configurations from JSON or YAML files, nor did they support incremental configuration, which is needed for specifying logger options from a command line.

To support a more flexible style, the module now offers `logging.config.dictConfig()` for specifying logging configuration with plain Python dictionaries. The configuration options include formatters, handlers, filters, and loggers. Here's a working example of a configuration dictionary:

```
{  
  "version": 1, 
  "formatters": {"brief": {"format": "%(levelname)-8s"}, 
                  "full": {"format": "%(asctime)s %(name)-15s"}}, 
  "handlers": {"console": { 
                  "class": "logging.StreamHandler", 
                  "formatter": "brief", 
                  "level": "INFO", 
                  "stream": "ext://sys.stdout"}, 
               "console_priority": { 
                  "class": "logging.StreamHandler", 
                  "formatter": "full", 
                  "level": "ERROR", 
                  "stream": "ext://sys.stderr"}}, 
  "root": {"level": "DEBUG", 
            "handlers": ["console", 
```
If that dictionary is stored in a file called `conf.json`, it can be loaded and called with code like this:

```python
>>> import json, logging.config
>>> with open('conf.json') as f:
...    conf = json.load(f)
...
>>> logging.config.dictConfig(conf)
>>> logging.info("Transaction completed normally")
INFO : root : Transaction completed normally
>>> logging.critical("Abnormal termination")
2011-02-17 11:14:36,694 root CRITICAL Abnormal termination
```

See also:

**PEP 391 - Dictionary Based Configuration for Logging**

PEP written by Vinay Sajip.
PEP 3148: The concurrent.futures module

Code for creating and managing concurrency is being collected in a new top-level namespace, `concurrent`. Its first member is a `futures` package which provides a uniform high-level interface for managing threads and processes.

The design for `concurrent.futures` was inspired by the `java.util.concurrent` package. In that model, a running call and its result are represented by a `Future` object that abstracts features common to threads, processes, and remote procedure calls. That object supports status checks (running or done), timeouts, cancellations, adding callbacks, and access to results or exceptions.

The primary offering of the new module is a pair of executor classes for launching and managing calls. The goal of the executors is to make it easier to use existing tools for making parallel calls. They save the effort needed to setup a pool of resources, launch the calls, create a results queue, add time-out handling, and limit the total number of threads, processes, or remote procedure calls.

Ideally, each application should share a single executor across multiple components so that process and thread limits can be centrally managed. This solves the design challenge that arises when each component has its own competing strategy for resource management.

Both classes share a common interface with three methods: `submit()` for scheduling a callable and returning a `Future` object; `map()` for scheduling many asynchronous calls at a time, and `shutdown()` for freeing resources. The class is a `context manager`
and can be used in a `with` statement to assure that resources are automatically released when currently pending futures are done executing.

A simple of example of `ThreadPoolExecutor` is a launch of four parallel threads for copying files:

```python
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest4.txt')
```

See also:

**PEP 3148 - Futures – Execute Computations Asynchronously**

PEP written by Brian Quinlan.

**Code for Threaded Parallel URL reads**, an example using threads to fetch multiple web pages in parallel.

**Code for computing prime numbers in parallel**, an example demonstrating `ProcessPoolExecutor`. 
PEP 3147: PYC Repository Directories

Python’s scheme for caching bytecode in .pyc files did not work well in environments with multiple Python interpreters. If one interpreter encountered a cached file created by another interpreter, it would recompile the source and overwrite the cached file, thus losing the benefits of caching.

The issue of “pyc fights” has become more pronounced as it has become commonplace for Linux distributions to ship with multiple versions of Python. These conflicts also arise with CPython alternatives such as Unladen Swallow.

To solve this problem, Python’s import machinery has been extended to use distinct filenames for each interpreter. Instead of Python 3.2 and Python 3.3 and Unladen Swallow each competing for a file called “mymodule.pyc”, they will now look for “mymodule.cpython-32.pyc”, “mymodule.cpython-33.pyc”, and “mymodule.unladen10.pyc”. And to prevent all of these new files from cluttering source directories, the pyc files are now collected in a “__pycache__” directory stored under the package directory.

Aside from the filenames and target directories, the new scheme has a few aspects that are visible to the programmer:

- Imported modules now have a __cached__ attribute which stores the name of the actual file that was imported:

```python
>>> import collections
>>> collections.__cached__ # doctest: +SKIP
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```
- The tag that is unique to each interpreter is accessible from the `imp` module:

```python
>>> import imp
>>> imp.get_tag()  # doctest: +SKIP
'cpython-32'
```

- Scripts that try to deduce source filename from the imported file now need to be smarter. It is no longer sufficient to simply strip the “c” from a “.pyc” filename. Instead, use the new functions in the `imp` module:

```python
>>> imp.source_from_cache('c:/py32/lib/__pycache__/collections.cpython-32.pyc', 'c:/py32/lib/collections.py')
>>> imp.cache_from_source('c:/py32/lib/collections.py', 'c:/py32/lib/__pycache__/collections.cpython-32.pyc')
```

- The `py_compile` and `compileall` modules have been updated to reflect the new naming convention and target directory. The command-line invocation of `compileall` has new options: `-i` for specifying a list of files and directories to compile and `-b` which causes bytecode files to be written to their legacy location rather than `__pycache__`.

- The `importlib.abc` module has been updated with new abstract base classes for loading bytecode files. The obsolete ABCs, PyLoader and PyPycLoader, have been deprecated (instructions on how to stay Python 3.1 compatible are included with the documentation).

**See also:**

PEP 3147 - PYC Repository Directories
PEP written by Barry Warsaw.
PEP 3149: ABI Version Tagged .so Files

The PYC repository directory allows multiple bytecode cache files to be co-located. This PEP implements a similar mechanism for shared object files by giving them a common directory and distinct names for each version.

The common directory is “pyshared” and the file names are made distinct by identifying the Python implementation (such as CPython, PyPy, Jython, etc.), the major and minor version numbers, and optional build flags (such as “d” for debug, “m” for pymalloc, “u” for wide-unicode). For an arbitrary package “foo”, you may see these files when the distribution package is installed:

```
/usr/share/pyshared/foo.cpython-32m.so
/usr/share/pyshared/foo.cpython-33md.so
```

In Python itself, the tags are accessible from functions in the `sysconfig` module:

```
>>> import sysconfig
>>> sysconfig.get_config_var('SOABI') # find the version tag
'cpython-32mu'
>>> sysconfig.get_config_var('EXT_SUFFIX') # find the full filename extension
'.cpython-32mu.so'
```

See also:

**PEP 3149 - ABI Version Tagged .so Files**

PEP written by Barry Warsaw.
PEP 3333: Python Web Server Gateway Interface v1.0.1

This informational PEP clarifies how bytes/text issues are to be handled by the WSGI protocol. The challenge is that string handling in Python 3 is most conveniently handled with the `str` type even though the HTTP protocol is itself bytes oriented.

The PEP differentiates so-called `native strings` that are used for request/response headers and metadata versus `byte strings` which are used for the bodies of requests and responses.

The `native strings` are always of type `str` but are restricted to code points between $U+0000$ through $U+00FF$ which are translatable to bytes using `Latin-1` encoding. These strings are used for the keys and values in the environment dictionary and for response headers and statuses in the `start_response()` function. They must follow `RFC 2616` with respect to encoding. That is, they must either be `ISO-8859-1` characters or use `RFC 2047` MIME encoding.

For developers porting WSGI applications from Python 2, here are the salient points:

- If the app already used strings for headers in Python 2, no change is needed.
- If instead, the app encoded output headers or decoded input headers, then the headers will need to be re-encoded to Latin-1. For example, an output header encoded in utf-8 was using `h.encode('utf-8')` now needs to convert from bytes to native strings using `h.encode('utf-8').decode('latin-1')`.
- Values yielded by an application or sent using the `write()` method must be byte strings. The `start_response()` function
and environ must use native strings. The two cannot be mixed.

For server implementers writing CGI-to-WSGI pathways or other CGI-style protocols, the users must to be able access the environment using native strings even though the underlying platform may have a different convention. To bridge this gap, the wsgiref module has a new function, wsgiref.handlers.read_environ() for transcoding CGI variables from os.environ into native strings and returning a new dictionary.

**See also:**

PEP 3333 - Python Web Server Gateway Interface v1.0.1
PEP written by Phillip Eby.
Other Language Changes

Some smaller changes made to the core Python language are:

- String formatting for `format()` and `str.format()` gained new capabilities for the format character `. Previously, for integers in binary, octal, or hexadecimal, it caused the output to be prefixed with ‘0b’, ‘0o’, or ‘0x’ respectively. Now it can also handle floats, complex, and Decimal, causing the output to always have a decimal point even when no digits follow it.

  ```python
  >>> format(20, '#o')
  '0o24'
  >>> format(12.34, '#5.0f')
  '12.'
  ```

  (Suggested by Mark Dickinson and implemented by Eric Smith in bpo-7094.)

- There is also a new `str.format_map()` method that extends the capabilities of the existing `str.format()` method by accepting arbitrary mapping objects. This new method makes it possible to use string formatting with any of Python’s many dictionary-like objects such as `defaultdict`, `Shelf`, `ConfigParser`, or `dbm`. It is also useful with custom `dict` subclasses that normalize keys before look-up or that supply a `__missing__()` method for unknown keys:

  ```python
  >>> import shelve
  >>> d = shelve.open('tmp.shl')
  >>> 'The {project_name} status is {status} as of February 15, 2011'
  'The testing project status is green as of February
  ```
>>> class LowerCasedDict(dict):
...     def __getitem__(self, key):
...         return dict.__getitem__(self, key.lower)
>>> lcd = LowerCasedDict(part='widgets', quantity=10)
>>> 'There are {QUANTITY} {Part} in stock'.format_map(lcd)
'There are 10 widgets in stock'

>>> class PlaceHolderDict(dict):
...     def __missing__(self, key):
...         return '<{key}>'.format(key=key)
>>> 'Hello {name}, welcome to {location}'.format_map(PlaceHolderDict())
'Hello <name>, welcome to <location>'

(Suggested by Raymond Hettinger and implemented by Eric Smith in bpo-6081.)

- The interpreter can now be started with a quiet option, `-q`, to prevent the copyright and version information from being displayed in the interactive mode. The option can be introspected using the `sys.flags` attribute:

```
$ python -q
>>> sys.flags
dsys.flags(debug=0, division_warning=0, inspect=0, optimize=0, dont_write_bytecode=0, no_user_site=0, ignore_environment=0, verbose=0, bytes_warning=0, quiet=1)
```

(Contributed by Marcin Wojdyr in bpo-1772833).

- The `hasattr()` function works by calling `getattr()` and detecting whether an exception is raised. This technique allows it to detect methods created dynamically by `__getattr__()` or `__getattribute__()` which would otherwise be absent from
the class dictionary. Formerly, `hasattr` would catch any exception, possibly masking genuine errors. Now, `hasattr` has been tightened to only catch `AttributeError` and let other exceptions pass through:

```python
>>> class A:
...     @property
...     def f(self):
...         return 1 // 0
...
>>> a = A()
>>> hasattr(a, 'f')
Traceback (most recent call last):
  ...ZeroDivisionError: integer division or modulo by zero
```

(Discovered by Yury Selivanov and fixed by Benjamin Peterson; bpo-9666.)

- The `str()` of a float or complex number is now the same as its `repr()`. Previously, the `str()` form was shorter but that just caused confusion and is no longer needed now that the shortest possible `repr()` is displayed by default:

```python
>>> import math
>>> repr(math.pi)
'3.141592653589793'
>>> str(math.pi)
'3.141592653589793'
```

(Proposed and implemented by Mark Dickinson; bpo-9337.)

- `memoryview` objects now have a `release()` method and they also now support the context management protocol. This allows
timely release of any resources that were acquired when requesting a buffer from the original object.

```python
>>> with memoryview(b'abcdefgh') as v:
...     print(v.tolist())
[97, 98, 99, 100, 101, 102, 103, 104]
```

(Added by Antoine Pitrou; bpo-9757.)

- Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block:

```python
def outer(x):
    def inner():
        return x
    inner()
    del x
```

This is now allowed. Remember that the target of an `except` clause is cleared, so this code which used to work with Python 2.6, raised a `SyntaxError` with Python 3.1 and now works again:

```python
def f():
    def print_error():
        print(e)
    try:
        something
    except Exception as e:
        print_error()
        # implicit "del e" here
```

(See bpo-4617.)

- The internal structsequence tool now creates subclasses of
tuple. This means that C structures like those returned by `os.stat()`, `time.gmtime()`, and `sys.version_info` now work like a named tuple and now work with functions and methods that expect a tuple as an argument. This is a big step forward in making the C structures as flexible as their pure Python counterparts:

```python
>>> import sys
>>> isinstance(sys.version_info, tuple)
True
>>> 'Version %d.%d.%d %s(%d)' % sys.version_info
'Version 3.2.0 final(0)'
```

(Suggested by Arfrever Frehtes Taifersar Arahesis and implemented by Benjamin Peterson in [bpo-8413](https://bugs.python.org/issue8413).)

- Warnings are now easier to control using the `PYTHONWARNINGS` environment variable as an alternative to using `-W` at the command line:

```
$ export PYTHONWARNINGS='ignore::RuntimeWarning::,once::UnicodeWarning::'
```

(Suggested by Barry Warsaw and implemented by Philip Jenvey in [bpo-7301](https://bugs.python.org/issue7301).)

- A new warning category, `ResourceWarning`, has been added. It is emitted when potential issues with resource consumption or cleanup are detected. It is silenced by default in normal release builds but can be enabled through the means provided by the `warnings` module, or on the command line.

A `ResourceWarning` is issued at interpreter shutdown if the
gc.garbage list isn’t empty, and if gc.DEBUG_UNCOLLECTABLE is set, all uncollectable objects are printed. This is meant to make the programmer aware that their code contains object finalization issues.

A ResourceWarning is also issued when a file object is destroyed without having been explicitly closed. While the deallocator for such object ensures it closes the underlying operating system resource (usually, a file descriptor), the delay in deallocating the object could produce various issues, especially under Windows. Here is an example of enabling the warning from the command line:

```
$ python -q -Wdefault
>>> f = open("foo", "wb")
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.BufferedWriter name='foo'>
```

(Added by Antoine Pitrou and Georg Brandl in bpo-10093 and bpo-477863.)

- range objects now support index and count methods. This is part of an effort to make more objects fully implement the collections.Sequence abstract base class. As a result, the language will have a more uniform API. In addition, range objects now support slicing and negative indices, even with values larger than sys.maxsize. This makes range more interoperable with lists:

```
>>> range(0, 100, 2).count(10)
1
>>> range(0, 100, 2).index(10)
5
```
The `callable()` built-in function from Py2.x was resurrected. It provides a concise, readable alternative to using an abstract base class in an expression like `isinstance(x, collections.Callable):

```python
>>> callable(max)
True
>>> callable(20)
False
```

(See bpo-10518.)

Python's import mechanism can now load modules installed in directories with non-ASCII characters in the path name. This solved an aggravating problem with home directories for users with non-ASCII characters in their usernames.

(Required extensive work by Victor Stinner in bpo-9425.)
New, Improved, and Deprecated Modules

Python’s standard library has undergone significant maintenance efforts and quality improvements.

The biggest news for Python 3.2 is that the email package, mailbox module, and nntplib modules now work correctly with the bytes/text model in Python 3. For the first time, there is correct handling of messages with mixed encodings.

Throughout the standard library, there has been more careful attention to encodings and text versus bytes issues. In particular, interactions with the operating system are now better able to exchange non-ASCII data using the Windows MBCS encoding, locale-aware encodings, or UTF-8.

Another significant win is the addition of substantially better support for SSL connections and security certificates.

In addition, more classes now implement a context manager to support convenient and reliable resource clean-up using a with statement.

email

The usability of the email package in Python 3 has been mostly fixed by the extensive efforts of R. David Murray. The problem was that emails are typically read and stored in the form of bytes rather than str text, and they may contain multiple encodings within a single email. So, the email package had to be extended to parse and generate email messages in bytes format.
- New functions `message_from_bytes()` and `message_from_binary_file()`, and new classes `BytesFeedParser` and `BytesParser` allow binary message data to be parsed into model objects.

- Given bytes input to the model, `get_payload()` will by default decode a message body that has a `Content-Transfer-Encoding` of `8bit` using the charset specified in the MIME headers and return the resulting string.

- Given bytes input to the model, `Generator` will convert message bodies that have a `Content-Transfer-Encoding` of `8bit` to instead have a `7bit Content-Transfer-Encoding`.

Headers with unencoded non-ASCII bytes are deemed to be RFC 2047-encoded using the `unknown-8bit` character set.

- A new class `BytesGenerator` produces bytes as output, preserving any unchanged non-ASCII data that was present in the input used to build the model, including message bodies with a `Content-Transfer-Encoding` of `8bit`.

- The `smtplib` SMTP class now accepts a byte string for the `msg` argument to the `sendmail()` method, and a new method, `send_message()` accepts a `Message` object and can optionally obtain the `from_addr` and `to_addrs` addresses directly from the object.

(Proposed and implemented by R. David Murray, bpo-4661 and bpo-10321.)

`elementtree`
The `xml.etree.ElementTree` package and its `xml.etree.cElementTree` counterpart have been updated to version 1.3.

Several new and useful functions and methods have been added:

- `xml.etree.ElementTree.fromstringlist()` which builds an XML document from a sequence of fragments
- `xml.etree.ElementTree.register_namespace()` for registering a global namespace prefix
- `xml.etree.ElementTree.tostringlist()` for string representation including all sublists
- `xml.etree.ElementTree.Element.extend()` for appending a sequence of zero or more elements
- `xml.etree.ElementTree.Element.iterfind()` searches an element and subelements
- `xml.etree.ElementTree.Element.itertext()` creates a text iterator over an element and its subelements
- `xml.etree.ElementTree.TreeBuilder.end()` closes the current element
- `xml.etree.ElementTree.TreeBuilder.doctype()` handles a doctype declaration

Two methods have been deprecated:

- `xml.etree.ElementTree.getchildren()` use `list(elem)` instead.
- `xml.etree.ElementTree.getiterator()` use `Element.iter` instead.

For details of the update, see Introducing `ElementTree` on Fredrik Lundh’s website.

(Contributed by Florent Xicluna and Fredrik Lundh, bpo-6472.)
functools

- The `functools` module includes a new decorator for caching function calls. `functools.lru_cache()` can save repeated queries to an external resource whenever the results are expected to be the same.

For example, adding a caching decorator to a database query function can save database accesses for popular searches:

```python
>>> import functools
>>> @functools.lru_cache(maxsize=300)
... def get_phone_number(name):
...     c = conn.cursor()
...     c.execute('SELECT phonenumber FROM phonelist WHERE name=?
...     return c.fetchone()[0]
```

```python
>>> for name in user_requests:  # doctest: +SKIP
...     get_phone_number(name)  # cached lookup
```

To help with choosing an effective cache size, the wrapped function is instrumented for tracking cache statistics:

```python
>>> get_phone_number.cache_info()  # doctest: +SKIP
CacheInfo(hits=4805, misses=980, maxsize=300, currsize=300)
```

If the phonelist table gets updated, the outdated contents of the cache can be cleared with:

```python
>>> get_phone_number.cache_clear()
```
The `functools.wraps()` decorator now adds a `__wrapped__` attribute pointing to the original callable function. This allows wrapped functions to be introspected. It also copies `__annotations__` if defined. And now it also gracefully skips over missing attributes such as `__doc__` which might not be defined for the wrapped callable.

In the above example, the cache can be removed by recovering the original function:

```python
>>> get_phone_number = get_phone_number.__wrapped__
```

(By Nick Coghlan and Terrence Cole; `bpo-9567`, `bpo-3445`, and `bpo-8814`.)

To help write classes with rich comparison methods, a new decorator `functools.total_ordering()` will use existing equality and inequality methods to fill in the remaining methods.

For example, supplying `__eq__` and `__lt__` will enable `total_ordering()` to fill-in `__le__`, `__gt__` and `__ge__`:

```python
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower(),
                 self.age),
                (other.lastname.lower(), other.firstname.lower(),
                 other.age))

    def __lt__(self, other):
        ```
With the *total_ordering* decorator, the remaining comparison methods are filled in automatically.

(Contributed by Raymond Hettinger.)

- To aid in porting programs from Python 2, the `functools.cmp_to_key()` function converts an old-style comparison function to modern key function:

```
>>> # locale-aware sort order
>>> sorted(iterable, key=cmp_to_key(locale.strcoll))
```

For sorting examples and a brief sorting tutorial, see the Sorting HowTo tutorial.

(Contributed by Raymond Hettinger.)

**itertools**

- The *itertools* module has a new *accumulate()* function modeled on APL’s *scan* operator and Numpy’s *accumulate* function:

```
>>> from itertools import accumulate
>>> list(accumulate([8, 2, 50]))
[8, 10, 60]
```

```
>>> prob_dist = [0.1, 0.4, 0.2, 0.3]
>>> list(accumulate(prob_dist))  # cumulative
[0.1, 0.5, 0.7, 1.0]
```
For an example using `accumulate()`, see the examples for the random module.

(Contributed by Raymond Hettinger and incorporating design suggestions from Mark Dickinson.)

collections

- The `collections.Counter` class now has two forms of in-place subtraction, the existing `-=` operator for **saturating subtraction** and the new `subtract()` method for regular subtraction. The former is suitable for **multisets** which only have positive counts, and the latter is more suitable for use cases that allow negative counts:

  ```python
  >>> from collections import Counter
  >>> tally = Counter(dogs=5, cats=3)
  >>> tally -= Counter(dogs=2, cats=8)  # saturating subtraction
  >>> tally
  Counter({'dogs': 3})
  >>> tally.subtract(dogs=2, cats=8)  # regular subtraction
  >>> tally
  Counter({'dogs': 3, 'cats': -5})
  ```

  (Contributed by Raymond Hettinger.)

- The `collections.OrderedDict` class has a new method `move_to_end()` which takes an existing key and moves it to
either the first or last position in the ordered sequence.

The default is to move an item to the last position. This is equivalent of renewing an entry with \( \text{od}[k] = \text{od}.\text{pop}(k) \).

A fast move-to-end operation is useful for resequencing entries. For example, an ordered dictionary can be used to track order of access by aging entries from the oldest to the most recently accessed.

```python
>>> from collections import OrderedDict
>>> d = OrderedDict.fromkeys(['a', 'b', 'X', 'd'])
>>> list(d)
['a', 'b', 'X', 'd', 'e']
>>> d.move_to_end('X')
>>> list(d)
['a', 'b', 'd', 'e', 'X']
```

(Contributed by Raymond Hettinger.)

- The `collections.deque` class grew two new methods `count()` and `reverse()` that make them more substitutable for `list` objects:

```python
>>> from collections import deque
>>> d = deque('simsalabim')
>>> d.count('s')
2
>>> d.reverse()
>>> d
deque(['m', 'i', 'b', 'a', 'l', 'a', 's', 'm', 'i', 's'])
```

(Contributed by Raymond Hettinger.)
threading

The `threading` module has a new `Barrier` synchronization class for making multiple threads wait until all of them have reached a common barrier point. Barriers are useful for making sure that a task with multiple preconditions does not run until all of the predecessor tasks are complete.

Barriers can work with an arbitrary number of threads. This is a generalization of a `Rendezvous` which is defined for only two threads.

Implemented as a two-phase cyclic barrier, `Barrier` objects are suitable for use in loops. The separate `filling` and `draining` phases assure that all threads get released (drained) before any one of them can loop back and re-enter the barrier. The barrier fully resets after each cycle.

Example of using barriers:

```python
from threading import Barrier, Thread

def get_votes(site):
    ballots = conduct_election(site)
    all_polls_closed.wait()  # do not count until all polls are closed
    totals = summarize(ballots)
    publish(site, totals)

all_polls_closed = Barrier(len(sites))
for site in sites:
    Thread(target=get_votes, args=(site,)).start()
```

In this example, the barrier enforces a rule that votes cannot be counted at any polling site until all polls are closed. Notice how a
solution with a barrier is similar to one with `threading.Thread.join()`, but the threads stay alive and continue to do work (summarizing ballots) after the barrier point is crossed.

If any of the predecessor tasks can hang or be delayed, a barrier can be created with an optional `timeout` parameter. Then if the timeout period elapses before all the predecessor tasks reach the barrier point, all waiting threads are released and a `BrokenBarrierError` exception is raised:

```python
def get_votes(site):
    ballots = conduct_election(site)
    try:
        all_polls_closed.wait(timeout=midnight - time)
    except BrokenBarrierError:
        lockbox = seal_ballots(ballots)
        queue.put(lockbox)
    else:
        totals = summarize(ballots)
publish(site, totals)
```

In this example, the barrier enforces a more robust rule. If some election sites do not finish before midnight, the barrier times-out and the ballots are sealed and deposited in a queue for later handling.

See `Barrier Synchronization Patterns` for more examples of how barriers can be used in parallel computing. Also, there is a simple but thorough explanation of barriers in `The Little Book of Semaphores`, section 3.6.

(Contributed by Kristján Valur Jónsson with an API review by Jeffrey Yasskin in `bpo-8777`.)
datetime and time

- The `datetime` module has a new type `timezone` that implements the `tzinfo` interface by returning a fixed UTC offset and timezone name. This makes it easier to create timezone-aware datetime objects:

```python
>>> from datetime import datetime, timezone
>>> datetime.now(timezone.utc)
datetime.datetime(2010, 12, 8, 21, 4, 2, 923754, tzinfo=datetime.timezone.utc)
>>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y %H:%M %Z")
datetime.datetime(2000, 1, 1, 12, 0, tzinfo=datetime.timezone.utc)
```

- Also, `timedelta` objects can now be multiplied by `float` and divided by `float` and `int` objects. And `timedelta` objects can now divide one another.

- The `datetime.date.strftime()` method is no longer restricted to years after 1900. The new supported year range is from 1000 to 9999 inclusive.

- Whenever a two-digit year is used in a time tuple, the interpretation has been governed by `time.accept2dyear`. The default is `True` which means that for a two-digit year, the century is guessed according to the POSIX rules governing the `%y` `strptime` format.

  Starting with Py3.2, use of the century guessing heuristic will emit a `DeprecationWarning`. Instead, it is recommended that `time.accept2dyear` be set to `False` so that large date ranges
can be used without guesswork:

```python
>>> import time, warnings
>>> warnings.resetwarnings()    # remove the default warning filters

>>> time.accept2dyear = True    # guess whether 11 means 11 or 2011
>>> time.asctime(((11, 1, 1, 12, 34, 56, 4, 1, 0)))

Warning (from warnings module):
... 
DeprecationWarning: Century info guessed for a 2-digit year.
'Fri Jan  1 12:34:56 2011'

>>> time.accept2dyear = False  # use the full range of allowble dates
>>> time.asctime(((11, 1, 1, 12, 34, 56, 4, 1, 0)))
'Fri Jan  1 12:34:56 11'
```

Several functions now have significantly expanded date ranges. When `time.accept2dyear` is false, the `time.asctime()` function will accept any year that fits in a C `int`, while the `time.mktime()` and `time.strftime()` functions will accept the full range supported by the corresponding operating system functions.

(Contributed by Alexander Belopolsky and Victor Stinner in bpo-1289118, bpo-5094, bpo-6641, bpo-2706, bpo-1777412, bpo-8013, and bpo-10827.)

**math**

The `math` module has been updated with six new functions inspired by the C99 standard.

The `isfinite()` function provides a reliable and fast way to detect
special values. It returns True for regular numbers and False for Nan or Infinity:

```python
>>> from math import isfinite
>>> [isfinite(x) for x in (123, 4.56, float('Nan')),
   [True, True, False, False]]
[True, True, False, False]
```

The expm1() function computes \( e^x - 1 \) for small values of \( x \) without incurring the loss of precision that usually accompanies the subtraction of nearly equal quantities:

```python
>>> from math import expm1
>>> expm1(0.013671875)  # more accurate way to compute e^x-1 for a small x
0.013765762467652909
```

The erf() function computes a probability integral or Gaussian error function. The complementary error function, erfc(), is \( 1 - \text{erf}(x) \):

```python
>>> from math import erf, erfc, sqrt
>>> erf(1.0/sqrt(2.0))  # portion of normal distribution
0.682689492137086
>>> erfc(1.0/sqrt(2.0))  # portion of normal distribution
0.31731050786291404
>>> erf(1.0/sqrt(2.0)) + erfc(1.0/sqrt(2.0))
1.0
```

The gamma() function is a continuous extension of the factorial function. See https://en.wikipedia.org/wiki/Gamma_function for details. Because the function is related to factorials, it grows large even for small values of \( x \), so there is also a lgamma() function for computing the natural logarithm of the gamma function:
```python
>>> from math import gamma, lgamma
>>> gamma(7.0)  # six factorial
720.0
>>> lgamma(801.0)  # log(800 factorial)
4551.950730698041
```

(Contributed by Mark Dickinson.)

**abc**

The `abc` module now supports `abstractclassmethod()` and `abstractstaticmethod()`. These tools make it possible to define an abstract base class that requires a particular `classmethod()` or `staticmethod()` to be implemented:

```python
class Temperature(metaclass=abc.ABCMeta):
    @abc.abstractclassmethod
    def from_fahrenheit(cls, t):
        ...  
    @abc.abstractclassmethod
    def from_celsius(cls, t):
        ...
```

(Patch submitted by Daniel Urban; [bpo-5867](https://bugs.python.org/issue5867).)

**io**

The `io.BytesIO` has a new method, `getbuffer()`, which provides functionality similar to `memoryview()`. It creates an editable view of the data without making a copy. The buffer’s random access and support for slice notation are well-suited to in-place editing:
>>> REC_LEN, LOC_START, LOC_LEN = 34, 7, 11

>>> def change_location(buffer, record_number, location):
...     start = record_number * REC_LEN + LOC_START
...     buffer[start: start+LOC_LEN] = location

>>> import io

>>> byte_stream = io.BytesIO(
...     b'G3805	storeroom	Main chassis  
...     b'X7899	shipping	Reserve cog  
...     b'L6988	receiving	Primary sprocket'
... )

>>> buffer = byte_stream.getbuffer()

>>> change_location(buffer, 1, b'warehouse  ')

>>> change_location(buffer, 0, b'showroom  ')

>>> print(byte_stream.getvalue())

b'G3805  showroom  Main chassis  '
b'X7899  warehouse  Reserve cog  '
b'L6988  receiving  Primary sprocket'

(Contributed by Antoine Pitrou in bpo-5506.)

reprlib

When writing a __repr__() method for a custom container, it is easy to forget to handle the case where a member refers back to the container itself. Python’s built-in objects such as list and set handle self-reference by displaying “...” in the recursive part of the representation string.

To help write such __repr__() methods, the reprlib module has a new decorator, recursive_repr(), for detecting recursive calls to __repr__() and substituting a placeholder string instead:
class MyList(list):
    ...
    @recursive_repr()
    def __repr__(self):
        return '<' + '|'.join(map(repr, self)) + '>

m = MyList('abc')
m.append(m)
m.append('x')
print(m)
<a'|'b'|'c'|...'|'x'>

(Contributed by Raymond Hettinger in bpo-9826 and bpo-9840.)

logging

In addition to dictionary-based configuration described above, the `logging` package has many other improvements.

The logging documentation has been augmented by a basic tutorial, an advanced tutorial, and a cookbook of logging recipes. These documents are the fastest way to learn about logging.

The `logging.basicConfig()` set-up function gained a `style` argument to support three different types of string formatting. It defaults to “%” for traditional %-formatting, can be set to “{” for the new `str.format()` style, or can be set to “$” for the shell-style formatting provided by `string.Template`. The following three configurations are equivalent:

```python
>>> from logging import basicConfig
>>> basicConfig(style='%', format='%(name)s -> %(levelname)s')
>>> basicConfig(style='{', format='{name} -> {levelname}')
```
If no configuration is set-up before a logging event occurs, there is now a default configuration using a StreamHandler directed to sys.stderr for events of WARNING level or higher. Formerly, an event occurring before a configuration was set-up would either raise an exception or silently drop the event depending on the value of logging.raiseExceptions. The new default handler is stored in logging.lastResort.

The use of filters has been simplified. Instead of creating a Filter object, the predicate can be any Python callable that returns True or False.

There were a number of other improvements that add flexibility and simplify configuration. See the module documentation for a full listing of changes in Python 3.2.

CSV

The csv module now supports a new dialect, unix_dialect, which applies quoting for all fields and a traditional Unix style with '\n' as the line terminator. The registered dialect name is unix.

The csv.DictWriter has a new method, writeheader() for writing-out an initial row to document the field names:

```python
>>> import csv, sys
>>> w = csv.DictWriter(sys.stdout, ['name', 'dept'],
                    "name","dept"
>>> w.writerows([}
(New dialect suggested by Jay Talbot in bpo-5975, and the new method suggested by Ed Abraham in bpo-1537721.)

contextlib

There is a new and slightly mind-blowing tool ContextDecorator that is helpful for creating a context manager that does double duty as a function decorator.

As a convenience, this new functionality is used by contextmanager() so that no extra effort is needed to support both roles.

The basic idea is that both context managers and function decorators can be used for pre-action and post-action wrappers. Context managers wrap a group of statements using a with statement, and function decorators wrap a group of statements enclosed in a function. So, occasionally there is a need to write a pre-action or post-action wrapper that can be used in either role.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, the contextmanager() provides both capabilities in a single definition:

```python
from contextlib import contextmanager
```
import logging

logging.basicConfig(level=logging.INFO)

@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: %s', name)
    yield
    logging.info('Exiting: %s', name)

Formerly, this would have only been usable as a context manager:

with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()

Now, it can be used as a decorator as well:

@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()

Trying to fulfill two roles at once places some limitations on the technique. Context managers normally have the flexibility to return an argument usable by a with statement, but there is no parallel for function decorators.

In the above example, there is not a clean way for the track_entry_and_exit context manager to return a logging instance for use in the body of enclosed statements.

(Contributed by Michael Foord in bpo-9110.)

decimal and fractions
Mark Dickinson crafted an elegant and efficient scheme for assuring that different numeric datatypes will have the same hash value whenever their actual values are equal (bpo-8188):

```python
assert hash(Fraction(3, 2)) == hash(1.5) == \
hash(Decimal("1.5")) == hash(complex(1.5, 0))
```

Some of the hashing details are exposed through a new attribute, `sys.hash_info`, which describes the bit width of the hash value, the prime modulus, the hash values for `infinity` and `nan`, and the multiplier used for the imaginary part of a number:

```python
>>> sys.hash_info # doctest: +SKIP
sys.hash_info(width=64, modulus=2305843009213693951,
```

An early decision to limit the inter-operability of various numeric types has been relaxed. It is still unsupported (and ill-advised) to have implicit mixing in arithmetic expressions such as `Decimal('1.1') + float('1.1')` because the latter loses information in the process of constructing the binary float. However, since existing floating point value can be converted losslessly to either a decimal or rational representation, it makes sense to add them to the constructor and to support mixed-type comparisons.

- The `decimal.Decimal` constructor now accepts `float` objects directly so there in no longer a need to use the `from_float()` method (bpo-8257).
- Mixed type comparisons are now fully supported so that `Decimal` objects can be directly compared with `float` and `fractions.Fraction` (bpo-2531 and bpo-8188).
Similar changes were made to `fractions.Fraction` so that the `from_float()` and `from_decimal()` methods are no longer needed (bpo-8294):

```python
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> Decimal(1.1)
Decimal('1.1000000000000000888178419700125232338905355443435558677700781372664171793125')
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

Another useful change for the `decimal` module is that the `Context.clamp` attribute is now public. This is useful in creating contexts that correspond to the decimal interchange formats specified in IEEE 754 (see bpo-8540).

(Contributed by Mark Dickinson and Raymond Hettinger.)

### ftp

The `ftplib.FTP` class now supports the context management protocol to unconditionally consume `socket.error` exceptions and to close the FTP connection when done:

```python
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()

'230 Anonymous login ok, restrictions apply.'
```

```
  dr-xr-xr-x 9 ftp   ftp  154 May  6 10:
  dr-xr-xr-x 9 ftp   ftp  154 May  6 10:
  dr-xr-xr-x 5 ftp   ftp  4096 May  6 10:
```
Other file-like objects such as `mmap.mmap` and `fileinput.input()` also grew auto-closing context managers:

```python
with fileinput.input(files=('log1.txt', 'log2.txt'))
    for line in f:
        process(line)
```

(Contributed by Tarek Ziadé and Giampaolo Rodolà in bpo-4972, and by Georg Brandl in bpo-8046 and bpo-1286.)

The `FTP_TLS` class now accepts a `context` parameter, which is a `ssl.SSLContext` object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; bpo-8806.)

**popen**

The `os.popen()` and `subprocess.Popen()` functions now support `with` statements for auto-closing of the file descriptors.

(Contributed by Antoine Pitrou and Brian Curtin in bpo-7461 and bpo-10554.)

**select**

The `select` module now exposes a new, constant attribute, `PIPE_BUF`, which gives the minimum number of bytes which are
guaranteed not to block when `select.select()` says a pipe is ready for writing.

```python
>>> import select
>>> select.PIPE_BUF  # doctest: +SKIP
512
```

(Available on Unix systems. Patch by Sébastien Sablé in `bpo-9862`)

gzip and zipfile

gzip.GzipFile now implements the `io.BufferedIOBase` abstract base class (except for `truncate()`). It also has a `peek()` method and supports unseekable as well as zero-padded file objects.

The `gzip` module also gains the `compress()` and `decompress()` functions for easier in-memory compression and decompression. Keep in mind that text needs to be encoded as `bytes` before compressing and decompressing:

```python
>>> import gzip
>>> s = 'Three shall be the number thou shalt count, 
>>> s += 'and the number of the counting shall be the 
>>> b = s.encode()  # convert
>>> len(b)
89
>>> c = gzip.compress(b)
>>> len(c)
77
>>> gzip.decompress(c).decode()[42]  # decompress
'Three shall be the number thou shalt count'
```

(Contributed by Anand B. Pillai in `bpo-3488`; and by Antoine Pitrou, Nir
Aides and Brian Curtin in bpo-9962, bpo-1675951, bpo-7471 and bpo-2846.)

Also, the zipfile.ZipExtFile class was reworked internally to represent files stored inside an archive. The new implementation is significantly faster and can be wrapped in an io.BufferedReader object for more speedups. It also solves an issue where interleaved calls to read and readline gave the wrong results.

(Patch submitted by Nir Aides in bpo-7610.)

tarfile

The TarFile class can now be used as a context manager. In addition, its add() method has a new option, filter, that controls which files are added to the archive and allows the file metadata to be edited.

The new filter option replaces the older, less flexible exclude parameter which is now deprecated. If specified, the optional filter parameter needs to be a keyword argument. The user-supplied filter function accepts a TarInfo object and returns an updated TarInfo object, or if it wants the file to be excluded, the function can return None:

```python
>>> import tarfile, glob

>>> def myfilter(tarinfo):
...     if tarinfo.isfile():  # only save real files
...         tarinfo.uname = 'monty'  # redact the user name
...     return tarinfo

>>> with tarfile.open(name='myarchive.tar.gz', mode='w') as archive:
...     for filename in glob.glob('*.txt'):
...         archive.add(filename, filter=myfilter)
```
The `hashlib` module has two new constant attributes listing the hashing algorithms guaranteed to be present in all implementations and those available on the current implementation:

```python
>>> import hashlib

>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}

>>> hashlib.algorithms_available
{'md2', 'SHA256', 'SHA512', 'dsaWithSHA', 'mdc2', 'SHA1', 'MD5', 'sha512', 'ripemd160', 'SHA1', 'MDC2', 'SHA', 'SHA384', 'ecdsa-with-SHA1', 'md4', 'md5', 'sha1', 'DSA-SHA', 'dsaEncryption', 'DSA', 'RIPEMD160', 'sha', 'MD5',
```

(Suggested by Carl Chenet in bpo-7418.)

The `ast` module has a wonderful a general-purpose tool for safely
evaluating expression strings using the Python literal syntax. The `ast.literal_eval()` function serves as a secure alternative to the builtin `eval()` function which is easily abused. Python 3.2 adds `bytes` and `set` literals to the list of supported types: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and `None`.

```python
>>> from ast import literal_eval

>>> request = "{'req': 3, 'func': 'pow', 'args': (2,
>>> literal_eval(request)
{'args': (2, 0.5), 'req': 3, 'func': 'pow'}

>>> request = "os.system('do something harmful')"
>>> literal_eval(request)
Traceback (most recent call last):
  ... 
ValueError: malformed node or string: <_ast.Call object at 0x101739a10>
```

(Implemented by Benjamin Peterson and Georg Brandl.)

**OS**

Different operating systems use various encodings for filenames and environment variables. The `os` module provides two new functions, `fsencode()` and `fsdecode()`, for encoding and decoding filenames:

```python
>>> import os
>>> filename = 'Sehenswürdigkeiten'
>>> os.fsencode(filename)
b'Sehensw\xc3\xbcrdigkeiten'
```

Some operating systems allow direct access to encoded bytes in the
environment. If so, the `os.supports_bytes_environ` constant will be true.

For direct access to encoded environment variables (if available), use the new `os.getenvb()` function or use `os.environb` which is a bytes version of `os.environ`.

(Contributed by Victor Stinner.)

**shutil**

The `shutil.copytree()` function has two new options:

- `ignore_dangling_symlinks`: when `symlinks=False` so that the function copies a file pointed to by a symlink, not the symlink itself. This option will silence the error raised if the file doesn’t exist.
- `copy_function`: is a callable that will be used to copy files. `shutil.copy2()` is used by default.

(Contributed by Tarek Ziadé.)

In addition, the `shutil` module now supports archiving operations for zipfiles, uncompressed tarfiles, gzipped tarfiles, and bzipped tarfiles. And there are functions for registering additional archiving file formats (such as xz compressed tarfiles or custom formats).

The principal functions are `make_archive()` and `unpack_archive()`. By default, both operate on the current directory (which can be set by `os.chdir()`) and on any sub-directories. The archive filename needs to be specified with a full pathname. The archiving step is non-destructive (the original files are left unchanged).
```python
>>> import shutil, pprint

>>> os.chdir('mydata')  # change to the source directory
>>> f = shutil.make_archive('/var/backup/mydata', 'zip')  # archive the current directory
>>> f                      # show the name of archive
'/var/backup/mydata.zip'
>>> os.chdir('tmp')       # change to an unpacking directory
>>> shutil.unpack_archive('/var/backup/mydata.zip')

>>> pprint.pprint(shutil.get_archive_formats())  # display known formats
[('bztar', 'bzip2'ed tar-file'),
 ('gztar', 'gzip'ed tar-file'),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

>>> shutil.register_archive_format(  # register a new archive format
...     name='xz',
...     function=xz.compress,  # callable archiving function
...     extra_args=[('level', 8)],  # arguments to the function
...     description='xz compression'
... )

(Contributed by Tarek Ziadé.)

sqlite3

The `sqlite3` module was updated to `pysqlite` version 2.6.0. It has two new capabilities.

- The `sqlite3.Connection.in_transit` attribute is true if there is an active transaction for uncommitted changes.
- The `sqlite3.Connection.enable_load_extension()` and `sqlite3.Connection.load_extension()` methods allow
you to load SQLite extensions from “.so” files. One well-known extension is the fulltext-search extension distributed with SQLite.

(Contributed by R. David Murray and Shashwat Anand; bpo-8845.)

html

A new html module was introduced with only a single function, escape(), which is used for escaping reserved characters from HTML markup:

```python
>>> import html
>>> html.escape('x > 2 && x < 7')
'x &gt; 2 &amp;&amp; x &lt; 7'
```

socket

The socket module has two new improvements.

- Socket objects now have a detach() method which puts the socket into closed state without actually closing the underlying file descriptor. The latter can then be reused for other purposes. (Added by Antoine Pitrou; bpo-8524.)
- socket.create_connection() now supports the context management protocol to unconditionally consume socket.error exceptions and to close the socket when done. (Contributed by Giampaolo Rodolà; bpo-9794.)

ssl

The ssl module added a number of features to satisfy common requirements for secure (encrypted, authenticated) internet
connections:

- A new class, `SSLContext`, serves as a container for persistent SSL data, such as protocol settings, certificates, private keys, and various other options. It includes a `wrap_socket()` for creating an SSL socket from an SSL context.
- A new function, `ssl.match_hostname()`, supports server identity verification for higher-level protocols by implementing the rules of HTTPS (from RFC 2818) which are also suitable for other protocols.
- The `ssl.wrap_socket()` constructor function now takes a `ciphers` argument. The `ciphers` string lists the allowed encryption algorithms using the format described in the OpenSSL documentation.
- When linked against recent versions of OpenSSL, the `ssl` module now supports the Server Name Indication extension to the TLS protocol, allowing multiple “virtual hosts” using different certificates on a single IP port. This extension is only supported in client mode, and is activated by passing the `server_hostname` argument to `ssl.SSLContext.wrap_socket()`.
- Various options have been added to the `ssl` module, such as `OP_NO_SSLv2` which disables the insecure and obsolete SSLv2 protocol.
- The extension now loads all the OpenSSL ciphers and digest algorithms. If some SSL certificates cannot be verified, they are reported as an “unknown algorithm” error.
- The version of OpenSSL being used is now accessible using the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer).

(Contributed by Antoine Pitrou in bpo-8850, bpo-1589, bpo-8322, bpo-5639, bpo-4870, bpo-8484, and bpo-8321.)
nntp

The `nntplib` module has a revamped implementation with better bytes and text semantics as well as more practical APIs. These improvements break compatibility with the nntplib version in Python 3.1, which was partly dysfunctional in itself.

Support for secure connections through both implicit (using `nntplib.NNTP_SSL`) and explicit (using `nntplib.NNTP.starttls()`) TLS has also been added.

(Contributed by Antoine Pitrou in bpo-9360 and Andrew Vant in bpo-1926.)

certificates

`http.client.HTTPSConnection`, `urllib.request.HTTPSHandler` and `urllib.request.urlopen()` now take optional arguments to allow for server certificate checking against a set of Certificate Authorities, as recommended in public uses of HTTPS.

(Added by Antoine Pitrou, bpo-9003.)

imaplib

Support for explicit TLS on standard IMAP4 connections has been added through the new `imaplib.IMAP4.starttls` method.

(Contributed by Lorenzo M. Catucci and Antoine Pitrou, bpo-4471.)
http.client

There were a number of small API improvements in the http.client module. The old-style HTTP 0.9 simple responses are no longer supported and the strict parameter is deprecated in all classes.

The HTTPConnection and HTTPSConnection classes now have a source_address parameter for a (host, port) tuple indicating where the HTTP connection is made from.

Support for certificate checking and HTTPS virtual hosts were added to HTTPSConnection.

The request() method on connection objects allowed an optional body argument so that a file object could be used to supply the content of the request. Conveniently, the body argument now also accepts an iterable object so long as it includes an explicit Content-Length header. This extended interface is much more flexible than before.

To establish an HTTPS connection through a proxy server, there is a new set_tunnel() method that sets the host and port for HTTP Connect tunneling.

To match the behavior of http.server, the HTTP client library now also encodes headers with ISO-8859-1 (Latin-1) encoding. It was already doing that for incoming headers, so now the behavior is consistent for both incoming and outgoing traffic. (See work by Armin Ronacher in bpo-10980.)

unittest
The unittest module has a number of improvements supporting test discovery for packages, easier experimentation at the interactive prompt, new testcase methods, improved diagnostic messages for test failures, and better method names.

- The command-line call `python -m unittest` can now accept file paths instead of module names for running specific tests (bpo-10620). The new test discovery can find tests within packages, locating any test importable from the top-level directory. The top-level directory can be specified with the `-t` option, a pattern for matching files with `-p`, and a directory to start discovery with `-s`:

  ```
  $ python -m unittest discover -s my_proj_dir -p .
  ```

  (Contributed by Michael Foord.)

- Experimentation at the interactive prompt is now easier because the `unittest.case.TestCase` class can now be instantiated without arguments:

  ```python
  >>> from unittest import TestCase
  >>> testCase().assertEqual(pow(2, 3), 8)
  ```

  (Contributed by Michael Foord.)

- The `unittest` module has two new methods, `assertWarns()` and `assertWarnsRegex()` to verify that a given warning type is triggered by the code under test:

  ```python
  with self.assertWarns(DeprecationWarning):
    legacy_function('XYZ')
  ```

  (Contributed by Antoine Pitrou, bpo-9754.)
Another new method, `assertCountEqual()` is used to compare two iterables to determine if their element counts are equal (whether the same elements are present with the same number of occurrences regardless of order):

```python
def test_anagram(self):
    self.assertCountEqual('algorithm', 'logarithm')
```

(Contributed by Raymond Hettinger.)

- A principal feature of the unittest module is an effort to produce meaningful diagnostics when a test fails. When possible, the failure is recorded along with a diff of the output. This is especially helpful for analyzing log files of failed test runs. However, since diffs can sometime be voluminous, there is a new `maxDiff` attribute that sets maximum length of diffs displayed.

- In addition, the method names in the module have undergone a number of clean-ups.

For example, `assertRegex()` is the new name for `assertRegexpMatches()` which was misnamed because the test uses `re.search()`, not `re.match()`. Other methods using regular expressions are now named using short form “Regex” in preference to “Regexp” – this matches the names used in other unittest implementations, matches Python's old name for the `re` module, and it has unambiguous camel-casing.

(Contributed by Raymond Hettinger and implemented by Ezio Melotti.)

- To improve consistency, some long-standing method aliases are being deprecated in favor of the preferred names:
<table>
<thead>
<tr>
<th>Old Name</th>
<th>Preferred Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert_()</td>
<td>assertTrue()</td>
</tr>
<tr>
<td>assertEquals()</td>
<td>assertEqual()</td>
</tr>
<tr>
<td>assertNotEquals()</td>
<td>assertNotEqual()</td>
</tr>
<tr>
<td>assertAlmostEquals()</td>
<td>assertAlmostEqual()</td>
</tr>
<tr>
<td>assertNotAlmostEquals()</td>
<td>assertNotAlmostEqual()</td>
</tr>
</tbody>
</table>

Likewise, the `TestCase.fail*` methods deprecated in Python 3.1 are expected to be removed in Python 3.3. Also see the Deprecated aliases section in the unittest documentation.

(Contributed by Ezio Melotti; bpo-9424.)

- The `assertDictContainsSubset()` method was deprecated because it was misimplemented with the arguments in the wrong order. This created hard-to-debug optical illusions where tests like `TestCase().assertDictContainsSubset({'a':1, 'b':2}, {'a':1})` would fail.

  (Contributed by Raymond Hettinger.)

**random**

The integer methods in the `random` module now do a better job of producing uniform distributions. Previously, they computed selections with `int(n*random())` which had a slight bias whenever \( n \) was not a power of two. Now, multiple selections are made from a range up to the next power of two and a selection is kept only when it falls within the range \( 0 \leq x < n \). The functions and methods affected are `randrange()`, `randint()`, `choice()`, `shuffle()` and
sample().

(Contributed by Raymond Hettinger; bpo-9025.)

**poplib**

**POP3_SSL** class now accepts a `context` parameter, which is a `ssl.SSLContext` object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; bpo-8807.)

**asyncore**

**asyncore.dispatcher** now provides a `handle_accepted()` method returning a `(sock, addr)` pair which is called when a connection has actually been established with a new remote endpoint. This is supposed to be used as a replacement for old `handle_accept()` and avoids the user to call `accept()` directly.

(Contributed by Giampaolo Rodolà; bpo-6706.)

**tempfile**

The **tempfile** module has a new context manager, **TemporaryDirectory** which provides easy deterministic cleanup of temporary directories:

```python
with tempfile.TemporaryDirectory() as tmpdirname:
    print('created temporary dir:', tmpdirname)
```
inspect

- The `inspect` module has a new function `getgeneratorstate()` to easily identify the current state of a generator-iterator:

```python
>>> from inspect import getgeneratorstate
>>> def gen():
...     yield 'demo'
>>> g = gen()
>>> getgeneratorstate(g)
'GEN_CREATED'
>>> next(g)
'demo'
>>> getgeneratorstate(g)
'GEN_SUSPENDED'
>>> next(g, None)
>>> getgeneratorstate(g)
'GEN_CLOSED'
```

- To support lookups without the possibility of activating a dynamic attribute, the `inspect` module has a new function, `getattr_static()`. Unlike `hasattr()`, this is a true read-only search, guaranteed not to change state while it is searching:

```python
>>> class A:
...     @property
...     def f(self):
...         print('Running')
```
... return 10
...

>>> a = A()
>>> getattr(a, 'f')
Running
10

>>> inspect.getattr_static(a, 'f')
<property object at 0x1022bd788>

(Contributed by Michael Foord.)

pydoc

The pydoc module now provides a much-improved Web server interface, as well as a new command-line option \texttt{-b} to automatically open a browser window to display that server:

$ pydoc3.2 -b

(Contributed by Ron Adam; \texttt{bpo-2001}.)

dis

The dis module gained two new functions for inspecting code, \texttt{code\_info()} and \texttt{show\_code()}. Both provide detailed code object information for the supplied function, method, source code string or code object. The former returns a string and the latter prints it:

```python
>>> import dis, random
>>> dis.show_code(random.choice)
Name: choice
Filename: /Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/random.py
Argument count: 2
```
In addition, the `dis()` function now accepts string arguments so that
the common idiom `dis(compile(s, '', 'eval'))` can be shortened to `dis(s):

```python
>>> dis('3*x+1 if x%2==1 else x//2')
1   0 LOAD_NAME 0 (x)
  3 LOAD_CONST 0 (2)
  6 BINARY_MODULO
  7 LOAD_CONST 1 (1)
 10 COMPARE_OP 2 (==)
13 POP_JUMP_IF_FALSE 28
 16 LOAD_CONST 2 (3)
 19 LOAD_NAME 0 (x)
 22 BINARY_MULTIPLY
 23 LOAD_CONST 1 (1)
 26 BINARY_ADD
 27 RETURN_VALUE
  28 LOAD_NAME 0 (x)
```
Taken together, these improvements make it easier to explore how CPython is implemented and to see for yourself what the language syntax does under-the-hood.

(Contributed by Nick Coghlan in bpo-9147.)

dbm

All database modules now support the `get()` and `setdefault()` methods.

(Suggested by Ray Allen in bpo-9523.)

c/types

A new type, `ctypes.c_ssize_t` represents the C `ssize_t` datatype.

site

The `site` module has three new functions useful for reporting on the details of a given Python installation.

- `getsitepackages()` lists all global site-packages directories.
- `getuserbase()` reports on the user’s base directory where data can be stored.
- `getusersitepackages()` reveals the user-specific site-packages directory path.
>>>
import site
>>>
site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages',
 '/Library/Frameworks/Python.framework/Versions/3.2/lib/site-python',
 '/Library/Python/3.2/site-packages']
>>>
site.getuserbase()
'/Users/raymondhettinger/Library/Python/3.2'
>>>
site.getusersitepackages()
'/Users/raymondhettinger/Library/Python/3.2/lib/python3.2/site-packages'

Conveniently, some of site's functionality is accessible directly from the command-line:

```
$ python -m site --user-base
/Users/raymondhettinger/.local
$ python -m site --user-site
/Users/raymondhettinger/.local/lib/python3.2/site-packages
```

(Contributed by Tarek Ziadé in bpo-6693.)

dsystinfo

The new `systinfo` module makes it straightforward to discover installation paths and configuration variables that vary across platforms and installations.

The module offers access simple access functions for platform and version information:

- `get_platform()` returning values like `linux-i586` or `macosx-10.6-ppc`.
- `get_python_version()` returns a Python version string such as “3.2”.
It also provides access to the paths and variables corresponding to one of seven named schemes used by distutils. Those include posix_prefix, posix_home, posix_user, nt, nt_user, os2, os2_home:

- get_paths() makes a dictionary containing installation paths for the current installation scheme.
- get_config_vars() returns a dictionary of platform specific variables.

There is also a convenient command-line interface:

```plaintext
C:\Python32>python -m sysconfig
Platform: "win32"
Python version: "3.2"
Current installation scheme: "nt"

Paths:
  data = "C:\Python32"
  include = "C:\Python32\Include"
  platinclude = "C:\Python32\Include"
  platlib = "C:\Python32\Lib\site-packages"
  platstdlib = "C:\Python32\Lib"
  purelib = "C:\Python32\Lib\site-packages"
  scripts = "C:\Python32\Scripts"
  stdlib = "C:\Python32\Lib"

Variables:
  BINDIR = "C:\Python32"
  BINLIBDEST = "C:\Python32\Lib"
  EXE = ".exe"
  INCLUDEPY = "C:\Python32\Include"
  LIBDEST = "C:\Python32\Lib"
  SO = ".pyd"
  VERSION = "32"
  abiflags = ""
  base = "C:\Python32"
```
```
exec_prefix = "C:\Python32"
platbase = "C:\Python32"
prefix = "C:\Python32"
projectbase = "C:\Python32"
py_version = "3.2"
py_version_nodot = "32"
py_version_short = "3.2"
srcdir = "C:\Python32"
userbase = "C:\Documents and Settings\Raymond"
```

(Moved out of Distutils by Tarek Ziadé.)

**pdb**

The **pdb** debugger module gained a number of usability improvements:

- `pdb.py` now has a `-c` option that executes commands as given in a `.pdbrc` script file.
- A `.pdbrc` script file can contain `continue` and `next` commands that continue debugging.
- The Pdb class constructor now accepts a `nosigint` argument.
- New commands: `l(list)`, `ll(long list)` and `source` for listing source code.
- New commands: `display` and `undisplay` for showing or hiding the value of an expression if it has changed.
- New command: `interact` for starting an interactive interpreter containing the global and local names found in the current scope.
- Breakpoints can be cleared by breakpoint number.

(Contributed by Georg Brandl, Antonio Cuni and Ilya Sandler.)

**configparser**
The `configparser` module was modified to improve usability and predictability of the default parser and its supported INI syntax. The old `ConfigParser` class was removed in favor of `SafeConfigParser` which has in turn been renamed to `ConfigParser`. Support for inline comments is now turned off by default and section or option duplicates are not allowed in a single configuration source.

Config parsers gained a new API based on the mapping protocol:

```python
>>> parser = ConfigParser()
>>> parser.read_string(""
... [DEFAULT]
... location = upper left
... visible = yes
... editable = no
... color = blue
...
... [main]
... title = Main Menu
... color = green
...
... [options]
... title = Options
... ""
>>> parser['main']['color']
'green'
>>> parser['main']['editable']
'no'
>>> section = parser['options']
>>> section['title']
'Options'
>>> section['title'] = 'Options (editable: %s)'
>>> section['title']
'Options (editable: no)'
```
The new API is implemented on top of the classical API, so custom parser subclasses should be able to use it without modifications.

The INI file structure accepted by config parsers can now be customized. Users can specify alternative option/value delimiters and comment prefixes, change the name of the DEFAULT section or switch the interpolation syntax.

There is support for pluggable interpolation including an additional interpolation handler **ExtendedInterpolation**:

```python
>>> parser = ConfigParser(interpolation=ExtendedInterpolation)
>>> parser.read_dict({'buildout': {'directory': '/home/ambv/zope9',
                                             'custom': {'prefix': '/usr/local'}}
>>> parser.read_string('"
... [buildout]
... parts =
...   zope9
... instance
... find-links =
...   ${buildout:directory}/downloads/dist
...
... [zope9]
... recipe = plone.recipe.zope9install
... location = /opt/zope
...
... [instance]
... recipe = plone.recipe.zope9instance
... zope9-location = ${zope9:location}
... zope-conf = ${custom:prefix}/etc/zope.conf
... ""

>>> parser['buildout']['find-links']
'
/home/ambv/zope9/downloads/dist'
>>> parser['instance']['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance = parser['instance']
```
A number of smaller features were also introduced, like support for specifying encoding in read operations, specifying fallback values for get-functions, or reading directly from dictionaries and strings.

(All changes contributed by Łukasz Langa.)

**urllib.parse**

A number of usability improvements were made for the *urllib.parse* module.

The *urlparse()* function now supports IPv6 addresses as described in RFC 2732:

```python
>>> import urllib.parse
ParseResult(scheme='http',
    path='/foo/',
    params='',
    query='',
    fragment='')
```

The *urldefrag()* function now returns a named tuple:

```python
>>> r = urllib.parse.urldefrag('http://python.org/about/#target')
>>> r
DefragResult(url='http://python.org/about/',
    fragment='target')
```
And, the `urlencode()` function is now much more flexible, accepting either a string or bytes type for the `query` argument. If it is a string, then the `safe`, `encoding`, and `error` parameters are sent to `quote_plus()` for encoding:

```
>>> urllib.parse.urlencode([
    ('type', 'telenovela'),
    ('name', '¿Dónde Está Elisa?'),
    encoding='latin-1')
'type=telenovela&name=%BFD%F3nde+Est%E1+Elisa%3F'
```

As detailed in Parsing ASCII Encoded Bytes, all the `urllib.parse` functions now accept ASCII-encoded byte strings as input, so long as they are not mixed with regular strings. If ASCII-encoded byte strings are given as parameters, the return types will also be an ASCII-encoded byte strings:

```
>>> urllib.parse.urlparse(b'http://www.python.org:80/about/
ParseResultBytes(scheme=b'http', netloc=b'www.python.org:80', path=b'/about/', params=b'', query=
```

(Work by Nick Coghlan, Dan Mahn, and Senthil Kumaran in `bpo-2987`, `bpo-5468`, and `bpo-9873`.)

**mailbox**

Thanks to a concerted effort by R. David Murray, the `mailbox` module
has been fixed for Python 3.2. The challenge was that mailbox had been originally designed with a text interface, but email messages are best represented with bytes because various parts of a message may have different encodings.

The solution harnessed the email package's binary support for parsing arbitrary email messages. In addition, the solution required a number of API changes.

As expected, the add() method for mailbox.Mailbox objects now accepts binary input.

StringIO and text file input are deprecated. Also, string input will fail early if non-ASCII characters are used. Previously it would fail when the email was processed in a later step.

There is also support for binary output. The get_file() method now returns a file in the binary mode (where it used to incorrectly set the file to text-mode). There is also a new get_bytes() method that returns a bytes representation of a message corresponding to a given key.

It is still possible to get non-binary output using the old API's get_string() method, but that approach is not very useful. Instead, it is best to extract messages from a Message object or to load them from binary input.

(Contributed by R. David Murray, with efforts from Steffen Daode Nurpmeso and an initial patch by Victor Stinner in bpo-9124.)

turtledemo
The demonstration code for the turtle module was moved from the Demo directory to main library. It includes over a dozen sample scripts with lively displays. Being on sys.path, it can now be run directly from the command-line:

$ python -m turtledemo

(Moved from the Demo directory by Alexander Belopolsky in bpo-10199.)
Multi-threading

- The mechanism for serializing execution of concurrently running Python threads (generally known as the GIL or Global Interpreter Lock) has been rewritten. Among the objectives were more predictable switching intervals and reduced overhead due to lock contention and the number of ensuing system calls. The notion of a “check interval” to allow thread switches has been abandoned and replaced by an absolute duration expressed in seconds. This parameter is tunable through `sys.setswitchinterval()`. It currently defaults to 5 milliseconds.

  Additional details about the implementation can be read from a python-dev mailing-list message (however, “priority requests” as exposed in this message have not been kept for inclusion).

  (Contributed by Antoine Pitrou.)

- Regular and recursive locks now accept an optional `timeout` argument to their `acquire()` method. (Contributed by Antoine Pitrou; bpo-7316.)

- Similarly, `threading.Semaphore.acquire()` also gained a `timeout` argument. (Contributed by Torsten Landschoff; bpo-850728.)

- Regular and recursive lock acquisitions can now be interrupted by signals on platforms using Pthreads. This means that Python programs that deadlock while acquiring locks can be successfully killed by repeatedly sending SIGINT to the process (by pressing Ctrl+C in most shells). (Contributed by Reid Kleckner; bpo-8844.)
Optimizations

A number of small performance enhancements have been added:

- Python’s peephole optimizer now recognizes patterns such as \( x \in \{1, 2, 3\} \) as being a test for membership in a set of constants. The optimizer recasts the \texttt{set} as a \texttt{frozenset} and stores the pre-built constant.

  Now that the speed penalty is gone, it is practical to start writing membership tests using set-notation. This style is both semantically clear and operationally fast:

  ```python
  extension = name.rpartition('.')[2]
  if extension in {'xml', 'html', 'xhtml', 'css'}:
      handle(name)
  ```

  (Patch and additional tests contributed by Dave Malcolm; \texttt{bpo-6690}).

- Serializing and unserializing data using the \texttt{pickle} module is now several times faster.

  (Contributed by Alexandre Vassalotti, Antoine Pitrou and the Unladen Swallow team in \texttt{bpo-9410} and \texttt{bpo-3873}).

- The \texttt{Timsort algorithm} used in \texttt{list.sort()} and \texttt{sorted()} now runs faster and uses less memory when called with a \texttt{key function}. Previously, every element of a list was wrapped with a temporary object that remembered the key value associated with each element. Now, two arrays of keys and values are sorted in parallel. This saves the memory consumed by the sort wrappers,
and it saves time lost to delegating comparisons.

(Patch by Daniel Stutzbach in bpo-9915.)

- JSON decoding performance is improved and memory consumption is reduced whenever the same string is repeated for multiple keys. Also, JSON encoding now uses the C speedups when the `sort_keys` argument is true.

(Contributed by Antoine Pitrou in bpo-7451 and by Raymond Hettinger and Antoine Pitrou in bpo-10314.)

- Recursive locks (created with the `threading.RLock()` API) now benefit from a C implementation which makes them as fast as regular locks, and between 10x and 15x faster than their previous pure Python implementation.

(Contributed by Antoine Pitrou; bpo-3001.)

- The fast-search algorithm in stringlib is now used by the `split()`, `rsplit()`, `splitlines()` and `replace()` methods on `bytes`, `bytearray` and `str` objects. Likewise, the algorithm is also used by `rfind()`, `rindex()`, `rsplit()` and `rpartition()`.

(Patch by Florent Xicluna in bpo-7622 and bpo-7462.)

- Integer to string conversions now work two “digits” at a time, reducing the number of division and modulo operations.

(bpo-6713 by Gawain Bolton, Mark Dickinson, and Victor Stinner.)

There were several other minor optimizations. Set differencing now runs faster when one operand is much larger than the other (patch by
Andress Bennetts in bpo-8685). The `array.repeat()` method has a faster implementation (bpo-1569291 by Alexander Belopolsky). The BaseHTTPRequestHandler has more efficient buffering (bpo-3709 by Andrew Schaaf). The `operator.attrgetter()` function has been sped-up (bpo-10160 by Christos Georgiou). And ConfigParser loads multi-line arguments a bit faster (bpo-7113 by Łukasz Langa).
Python has been updated to **Unicode 6.0.0**. The update to the standard adds over 2,000 new characters including **emoji** symbols which are important for mobile phones.

In addition, the updated standard has altered the character properties for two Kannada characters (U+0CF1, U+0CF2) and one New Tai Lue numeric character (U+19DA), making the former eligible for use in identifiers while disqualifying the latter. For more information, see [Unicode Character Database Changes](Unicode Character Database Changes).
Codecs

Support was added for cp720 Arabic DOS encoding ([bpo-1616979](https://bugs.python.org/issue1616979)).

MBCS encoding no longer ignores the error handler argument. In the default strict mode, it raises an `UnicodeDecodeError` when it encounters an undecodable byte sequence and an `UnicodeEncodeError` for an unencodable character.

The MBCS codec supports 'strict' and 'ignore' error handlers for decoding, and 'strict' and 'replace' for encoding.

To emulate Python3.1 MBCS encoding, select the 'ignore' handler for decoding and the 'replace' handler for encoding.

On Mac OS X, Python decodes command line arguments with 'utf-8' rather than the locale encoding.

By default, `tarfile` uses 'utf-8' encoding on Windows (instead of 'mbcs') and the 'surrogateescape' error handler on all operating systems.
Documentation

The documentation continues to be improved.

- A table of quick links has been added to the top of lengthy sections such as Built-in Functions. In the case of itertools, the links are accompanied by tables of cheatsheet-style summaries to provide an overview and memory jog without having to read all of the docs.

- In some cases, the pure Python source code can be a helpful adjunct to the documentation, so now many modules now feature quick links to the latest version of the source code. For example, the functools module documentation has a quick link at the top labeled:

  **Source code** Lib/functools.py.

  (Contributed by Raymond Hettinger; see rationale.)

- The docs now contain more examples and recipes. In particular, re module has an extensive section, Regular Expression Examples. Likewise, the itertools module continues to be updated with new Itertools Recipes.

- The datetime module now has an auxiliary implementation in pure Python. No functionality was changed. This just provides an easier-to-read alternate implementation.

  (Contributed by Alexander Belopolsky in bpo-9528.)

- The unmaintained Demo directory has been removed. Some demos were integrated into the documentation, some were moved
to the `Tools/demo` directory, and others were removed altogether.

(Contributed by Georg Brandl in `bpo-7962`.)
The format menu now has an option to clean source files by stripping trailing whitespace.

(Contributed by Raymond Hettinger; bpo-5150.)

IDLE on Mac OS X now works with both Carbon AquaTk and Cocoa AquaTk.

(Contributed by Kevin Walzer, Ned Deily, and Ronald Oussoren; bpo-6075.)
In addition to the existing Subversion code repository at http://svn.python.org there is now a Mercurial repository at https://hg.python.org/.

After the 3.2 release, there are plans to switch to Mercurial as the primary repository. This distributed version control system should make it easier for members of the community to create and share external changesets. See PEP 385 for details.

To learn to use the new version control system, see the tutorial by Joel Spolsky or the Guide to Mercurial Workflows.
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The `idle`, `pydoc` and `2to3` scripts are now installed with a version-specific suffix on make `altinstall` (bpo-10679).

- The C functions that access the Unicode Database now accept and return characters from the full Unicode range, even on narrow unicode builds (Py_UNICODE_TOLOWER, Py_UNICODE_ISDECIMAL, and others). A visible difference in Python is that `unicodedata.numeric()` now returns the correct value for large code points, and `repr()` may consider more characters as printable.

  (Reported by Bupjoe Lee and fixed by AmauryForgeot D’Arc; bpo-5127.)

- Computed gotos are now enabled by default on supported compilers (which are detected by the configure script). They can still be disabled selectively by specifying `--without-computed-gotos`.

  (Contributed by Antoine Pitrou; bpo-9203.)

- The option `--with-wctype-functions` was removed. The built-in unicode database is now used for all functions.

  (Contributed by AmauryForgeot D’Arc; bpo-9210.)

- Hash values are now values of a new type, `Py_hash_t`, which is defined to be the same size as a pointer. Previously they were of type `long`, which on some 64-bit operating systems is still only 32
bits long. As a result of this fix, **set** and **dict** can now hold more than \(2^{**32}\) entries on builds with 64-bit pointers (previously, they could grow to that size but their performance degraded catastrophically).

(Suggested by Raymond Hettinger and implemented by Benjamin Peterson; **bpo-9778**.)

- A new macro **Py VA COPY** copies the state of the variable argument list. It is equivalent to C99 **va_copy** but available on all Python platforms (**bpo-2443**).

- A new C API function **PySys_SetArgvEx()** allows an embedded interpreter to set **sys.argv** without also modifying **sys.path** (**bpo-5753**).

- **PyEval_CallObject** is now only available in macro form. The function declaration, which was kept for backwards compatibility reasons, is now removed – the macro was introduced in 1997 (**bpo-8276**).

- There is a new function **PyLong_AsLongLongAndOverflow()** which is analogous to **PyLong_AsLongAndOverflow()**. They both serve to convert Python **int** into a native fixed-width type while providing detection of cases where the conversion won’t fit (**bpo-7767**).

- The **PyUnicode_CompareWithASCIIString()** function now returns *not equal* if the Python string is *NUL* terminated.

- There is a new function **PyErr_NewExceptionWithDoc()** that is like **PyErr_NewException()** but allows a docstring to be specified. This lets C exceptions have the same self-documenting
capabilities as their pure Python counterparts (bpo-7033).

- When compiled with the `--with-valgrind` option, the pymalloc allocator will be automatically disabled when running under Valgrind. This gives improved memory leak detection when running under Valgrind, while taking advantage of pymalloc at other times (bpo-2422).

- Removed the `O?` format from the `PyArg_Parse` functions. The format is no longer used and it had never been documented (bpo-8837).

There were a number of other small changes to the C-API. See the Misc/NEWS file for a complete list.

Also, there were a number of updates to the Mac OS X build, see Mac/BuildScript/README.txt for details. For users running a 32/64-bit build, there is a known problem with the default Tcl/Tk on Mac OS X 10.6. Accordingly, we recommend installing an updated alternative such as ActiveState Tcl/Tk 8.5.9. See https://www.python.org/download/mac/tcltk/ for additional details.
Porting to Python 3.2

This section lists previously described changes and other bugfixes that may require changes to your code:

- The `configparser` module has a number of clean-ups. The major change is to replace the old `ConfigParser` class with long-standing preferred alternative `SafeConfigParser`. In addition there are a number of smaller incompatibilities:
  
  - The interpolation syntax is now validated on `get()` and `set()` operations. In the default interpolation scheme, only two tokens with percent signs are valid: `%{name}s` and `%%`, the latter being an escaped percent sign.
  
  - The `set()` and `add_section()` methods now verify that values are actual strings. Formerly, unsupported types could be introduced unintentionally.
  
  - Duplicate sections or options from a single source now raise either `DuplicateSectionError` or `DuplicateOptionError`. Formerly, duplicates would silently overwrite a previous entry.
  
  - Inline comments are now disabled by default so now the `;` character can be safely used in values.
  
  - Comments now can be indented. Consequently, for `;` or `#` to appear at the start of a line in multiline values, it has to be interpolated. This keeps comment prefix characters in values from being mistaken as comments.
  
  - `` is now a valid value and is no longer automatically converted to an empty string. For empty strings, use "option =" in a line.

- The `nntplib` module was reworked extensively, meaning that its APIs are often incompatible with the 3.1 APIs.
- `bytearray` objects can no longer be used as filenames; instead, they should be converted to `bytes`.

- The `array.tostring()` and `array.fromstring()` have been renamed to `array.tobytes()` and `array.frombytes()` for clarity. The old names have been deprecated. (See bpo-8990.)

- `PyArg_Parse*()` functions:
  - “t#” format has been removed: use “s#” or “s*” instead
  - “w” and “w#” formats has been removed: use “w*” instead

- The `PyCObject` type, deprecated in 3.1, has been removed. To wrap opaque C pointers in Python objects, the `PyCapsule` API should be used instead; the new type has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor.

- The `sys.setfilesystemencoding()` function was removed because it had a flawed design.

- The `random.seed()` function and method now salt string seeds with an sha512 hash function. To access the previous version of `seed` in order to reproduce Python 3.1 sequences, set the `version` argument to 1, `random.seed(s, version=1)`.

- The previously deprecated `string.maketrans()` function has been removed in favor of the static methods `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own `maketrans` and `translate` methods with intermediate translation tables of the appropriate type.
The previously deprecated `contextlib.nested()` function has been removed in favor of a plain `with` statement which can accept multiple context managers. The latter technique is faster (because it is built-in), and it does a better job finalizing multiple context managers when one of them raises an exception:

```python
with open('mylog.txt') as infile, open('a.out',
    for line in infile:
        if '<critical>' in line:
            outfile.write(line)
```

(Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094.)

`struct.pack()` now only allows bytes for the `s` string pack code. Formerly, it would accept text arguments and implicitly encode them to bytes using UTF-8. This was problematic because it made assumptions about the correct encoding and because a variable-length encoding can fail when writing to fixed length segment of a structure.

Code such as `struct.pack('<6sHHBBB', 'GIF87a', x, y)` should be rewritten with to use bytes instead of text, `struct.pack('<6sHHBBB', b'GIF87a', x, y)`.

(Discovered by David Beazley and fixed by Victor Stinner; bpo-10783.)

The `xml.etree.ElementTree` class now raises an `xml.etree.ElementTree.ParseError` when a parse fails.

(Contributed by Georg Brandl; bpo-5675.)
Previously it raised an `xml.parsers.expat.ExpatError`.

- The new, longer `str()` value on floats may break doctests which rely on the old output format.

- In `subprocess.Popen`, the default value for `close_fds` is now `True` under Unix; under Windows, it is `True` if the three standard streams are set to `None`, `False` otherwise. Previously, `close_fds` was always `False` by default, which produced difficult to solve bugs or race conditions when open file descriptors would leak into the child process.

- Support for legacy HTTP 0.9 has been removed from `urllib.request` and `http.client`. Such support is still present on the server side (in `http.server`).

  (Contributed by Antoine Pitrou, bpo-10711.)

- SSL sockets in timeout mode now raise `socket.timeout` when a timeout occurs, rather than a generic `SSLError`.

  (Contributed by Antoine Pitrou, bpo-10272.)

- The misleading functions `PyEval_AcquireLock()` and `PyEval_ReleaseLock()` have been officially deprecated. The thread-state aware APIs (such as `PyEval_SaveThread()` and `PyEval_RestoreThread()`) should be used instead.

- Due to security risks, `asyncore.handle_accept()` has been deprecated, and a new function, `asyncore.handle_accepted()`, was added to replace it.

  (Contributed by Giampaolo Rodola in bpo-6706.)
Due to the new GIL implementation, `PyEval_InitThreads()` cannot be called before `Py_Initialize()` anymore.
What’s New In Python 3.1

Author: Raymond Hettinger

This article explains the new features in Python 3.1, compared to 3.0.
PEP 372: Ordered Dictionaries

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, a new `collections.OrderedDict` class has been introduced.

The OrderedDict API is substantially the same as regular dictionaries but will iterate over keys and values in a guaranteed order depending on when a key was first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

The standard library now supports use of ordered dictionaries in several modules. The `configparser` module uses them by default. This lets configuration files be read, modified, and then written back in their original order. The `asdict()` method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices. The `json` module is being built-out with an `object_pairs_hook` to allow OrderedDicts to be built by the decoder. Support was also added for third-party tools like PyYAML.

See also:

**PEP 372 - Ordered Dictionaries**

PEP written by Armin Ronacher and Raymond Hettinger. Implementation written by Raymond Hettinger.
PEP 378: Format Specifier for Thousands Separator

The built-in `format()` function and the `str.format()` method use a mini-language that now includes a simple, non-locale aware way to format a number with a thousands separator. That provides a way to humanize a program's output, improving its professional appearance and readability:

```python
>>> format(1234567, ',d')
'1,234,567'
>>> format(1234567.89, ',.2f')
'1,234,567.89'
>>> format(12345.6 + 8901234.12j, ',f')
'12,345.600000+8,901.234.120000j'
>>> format(Decimal('1234567.89'), ',f')
'1,234,567.89'
```

The supported types are `int`, `float`, `complex` and `decimal.Decimal`.

Discussions are underway about how to specify alternative separators like dots, spaces, apostrophes, or underscores. Locale-aware applications should use the existing `n` format specifier which already has some support for thousands separators.

See also:

**PEP 378 - Format Specifier for Thousands Separator**  
PEP written by Raymond Hettinger and implemented by Eric Smith and Mark Dickinson.
Other Language Changes

Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory/zipfile is automatically inserted as the first entry in `sys.path`. (Suggestion and initial patch by Andy Chu; revised patch by Phillip J. Eby and Nick Coghlan; bpo-1739468.)

- The `int()` type gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

```
>>> n = 37
>>> bin(37)
'0b100101'
>>> n.bit_length()
6
>>> n = 2**123-1
>>> n.bit_length()
123
>>> (n+1).bit_length()
124
```

(Contributed by Fredrik Johansson, Victor Stinner, Raymond Hettinger, and Mark Dickinson; bpo-3439.)

- The fields in `format()` strings can now be automatically numbered:

```
>>> 'Sir {{}} of {{}}'.format('Gallahad', 'Camelot')
'Sir Gallahad of Camelot'
```
Formerly, the string would have required numbered fields such as:  
'Sir {0} of {1}'.

(Contributed by Eric Smith; bpo-5237.)

- The string.maketrans() function is deprecated and is replaced by new static methods, bytes.maketrans() and bytearray.maketrans(). This change solves the confusion around which types were supported by the string module. Now, str, bytes, and bytearray each have their own maketrans and translate methods with intermediate translation tables of the appropriate type.

  (Contributed by Georg Brandl; bpo-5675.)

- The syntax of the with statement now allows multiple context managers in a single statement:

```
>>> with open('mylog.txt') as infile, open('a.out') as outfile:
    for line in infile:
        if '<critical>' in line:
            outfile.write(line)
```

With the new syntax, the contextlib.nested() function is no longer needed and is now deprecated.

(Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094.)

- round(x, n) now returns an integer if x is an integer. Previously it returned a float:

```
>>> round(1123, -2)
```
Python now uses David Gay's algorithm for finding the shortest floating point representation that doesn't change its value. This should help mitigate some of the confusion surrounding binary floating point numbers.

The significance is easily seen with a number like 1.1 which does not have an exact equivalent in binary floating point. Since there is no exact equivalent, an expression like `float('1.1')` evaluates to the nearest representable value which is 0x1.199999999999ap+0 in hex or 1.10000000000000008881784197001252323389053344726 in decimal. That nearest value was and still is used in subsequent floating point calculations.

What is new is how the number gets displayed. Formerly, Python used a simple approach. The value of `repr(1.1)` was computed as `format(1.1, '.17g')` which evaluated to '1.1000000000000001'. The advantage of using 17 digits was that it relied on IEEE-754 guarantees to assure that `eval(repr(1.1))` would round-trip exactly to its original value. The disadvantage is that many people found the output to be confusing (mistaking intrinsic limitations of binary floating point representation as being a problem with Python itself).

The new algorithm for `repr(1.1)` is smarter and returns '1.1'. Effectively, it searches all equivalent string representations (ones that get stored with the same underlying float value) and returns the shortest representation.
The new algorithm tends to emit cleaner representations when possible, but it does not change the underlying values. So, it is still the case that $1.1 + 2.2 \neq 3.3$ even though the representations may suggest otherwise.

The new algorithm depends on certain features in the underlying floating point implementation. If the required features are not found, the old algorithm will continue to be used. Also, the text pickle protocols assure cross-platform portability by using the old algorithm.

(Contributed by Eric Smith and Mark Dickinson; bpo-1580)
New, Improved, and Deprecated Modules

- Added a `collections.Counter` class to support convenient counting of unique items in a sequence or iterable:

  ```python
  >>> Counter(['red', 'blue', 'red', 'green', 'blue'])
  Counter({'blue': 3, 'red': 2, 'green': 1})
  ```

  (Contributed by Raymond Hettinger; bpo-1696199.)

- Added a new module, `tkinter.ttk` for access to the Tk themed widget set. The basic idea of ttk is to separate, to the extent possible, the code implementing a widget’s behavior from the code implementing its appearance.

  (Contributed by Guilherme Polo; bpo-2983.)

- The `gzip.GzipFile` and `bz2.BZ2File` classes now support the context management protocol:

  ```python
  >>> # Automatically close file after writing
  >>> with gzip.GzipFile(filename, "wb") as f:
  ...     f.write(b"xxx")
  ```

  (Contributed by Antoine Pitrou.)

- The `decimal` module now supports methods for creating a decimal object from a binary `float`. The conversion is exact but can sometimes be surprising:

  ```python
  >>> Decimal.from_float(1.1)
  Decimal('1.100000000000000088817841970012523233890533447265625')
  ```
The long decimal result shows the actual binary fraction being stored for 1.1. The fraction has many digits because 1.1 cannot be exactly represented in binary.

(Contributed by Raymond Hettinger and Mark Dickinson.)

- The `itertools` module grew two new functions. The `itertools.combinations_with_replacement()` function is one of four for generating combinatorics including permutations and Cartesian products. The `itertools.compress()` function mimics its namesake from APL. Also, the existing `itertools.count()` function now has an optional `step` argument and can accept any type of counting sequence including `fractions.Fraction` and `decimal.Decimal`:

```python
>>> [p+q for p,q in combinations_with_replacement(['LL', 'L0', 'LV', 'LE', '0O', 'OV', 'OE', 'VV'], 2, 3, 5, 7]

>>> list(compress(data=range(10), selectors=[0, 0, 2, 3, 5, 7])

>>> c = count(start=Fraction(1,2), step=Fraction(1,2))

>>> [next(c), next(c), next(c), next(c), next(c)]

[Fraction(1, 2), Fraction(2, 3), Fraction(5, 6),

(Contributed by Raymond Hettinger.)

- `collections.namedtuple()` now supports a keyword argument `rename` which lets invalid fieldnames be automatically converted to positional names in the form _0, _1, etc. This is useful when the field names are being created by an external source such as a CSV header, SQL field list, or user input:
>>> query = input()
SELECT region, dept, count(*) FROM main GROUPBY region, dept

>>> cursor.execute(query)
>>> query_fields = [desc[0] for desc in cursor.description]
>>> UserQuery = namedtuple('UserQuery', query_fields)
>>> pprint.pprint([UserQuery(**row) for row in cursor])
[UserQuery(region='South', dept='Shipping', _2=185),
 UserQuery(region='North', dept='Accounting', _2=37),
 UserQuery(region='West', dept='Sales', _2=419)]

(Contributed by Raymond Hettinger; bpo-1818.)

- The `re.sub()`, `re.subn()` and `re.split()` functions now accept a flags parameter.

  (Contributed by Gregory Smith.)

- The `logging` module now implements a simple `logging.NullHandler` class for applications that are not using logging but are calling library code that does. Setting-up a null handler will suppress spurious warnings such as “No handlers could be found for logger foo”:

  ```
  >>> h = logging.NullHandler()
  >>> logging.getLogger("foo").addHandler(h)
  ```

  (Contributed by Vinay Sajip; bpo-4384.)

- The `runpy` module which supports the `-m` command line switch now supports the execution of packages by looking for and executing a `__main__` submodule when a package name is supplied.
- The **pdb** module can now access and display source code loaded via **zipimport** (or any other conformant **PEP 302** loader).

- **functools.partial** objects can now be pickled.

- Add **pydoc** help topics for symbols so that **help('@')** works as expected in the interactive environment.

- The **unittest** module now supports skipping individual tests or classes of tests. And it supports marking a test as an expected failure, a test that is known to be broken, but shouldn't be counted as a failure on a TestResult:

```python
class TestGizmo(unittest.TestCase):

    @unittest.skipUnless(sys.platform.startswith('win'),
    def test_gizmo_on_windows(self):
        ...

    @unittest.expectedFailure
    def test_gizmo_without_required_library(self):
        ...
```

Also, tests for exceptions have been builtout to work with context managers using the **with** statement:
```python
def test_division_by_zero(self):
    with self.assertRaises(ZeroDivisionError):
        x / 0
```

In addition, several new assertion methods were added including `assertSetEqual()`, `assertDictEqual()`, `assertDictContainsSubset()`, `assertListEqual()`, `assertTupleEqual()`, `assertSequenceEqual()`, `assertRaisesRegexp()`, `assertIsNone()`, and `assertIsNotNone()`.

(Contributed by Benjamin Peterson and Antoine Pitrou.)

- The `io` module has three new constants for the `seek()` method `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

- The `sys.version_info` tuple is now a named tuple:

  ```
  >>> sys.version_info
  sys.version_info(major=3, minor=1, micro=0, releaselevel='alpha', serial=2)
  ```

  (Contributed by Ross Light; bpo-4285.)

- The `nntplib` and `imaplib` modules now support IPv6.

  (Contributed by Derek Morr; bpo-1655 and bpo-1664.)

- The `pickle` module has been adapted for better interoperability with Python 2.x when used with protocol 2 or lower. The reorganization of the standard library changed the formal reference for many objects. For example, `__builtin__.set` in Python 2 is called `builtins.set` in Python 3. This change confounded efforts to share data between different versions of Python.
Python. But now when protocol 2 or lower is selected, the pickler will automatically use the old Python 2 names for both loading and dumping. This remapping is turned-on by default but can be disabled with the `fix_imports` option:

```python
>>> s = {1, 2, 3}
>>> pickle.dumps(s, protocol=0)
b'c__builtin__\nset\np0\\n((lp1\n1L\naL2L\naL3L\n
>>> pickle.dumps(s, protocol=0, fix_imports=False)
b'cbuiltins\nset\np0\\n((lp1\n1L\naL2L\naL3L\n
An unfortunate but unavoidable side-effect of this change is that protocol 2 pickles produced by Python 3.1 won’t be readable with Python 3.0. The latest pickle protocol, protocol 3, should be used when migrating data between Python 3.x implementations, as it doesn’t attempt to remain compatible with Python 2.x.

(Contributed by Alexandre Vassalotti and Antoine Pitrou, bpo-6137.)

- A new module, `importlib` was added. It provides a complete, portable, pure Python reference implementation of the `import` statement and its counterpart, the `__import__()` function. It represents a substantial step forward in documenting and defining the actions that take place during imports.

(Contributed by Brett Cannon.)
Optimizations

Major performance enhancements have been added:

- The new I/O library (as defined in PEP 3116) was mostly written in Python and quickly proved to be a problematic bottleneck in Python 3.0. In Python 3.1, the I/O library has been entirely rewritten in C and is 2 to 20 times faster depending on the task at hand. The pure Python version is still available for experimentation purposes through the _pyio module.

  (Contributed by Amaury Forgeot d'Arc and Antoine Pitrou.)

- Added a heuristic so that tuples and dicts containing only untrackable objects are not tracked by the garbage collector. This can reduce the size of collections and therefore the garbage collection overhead on long-running programs, depending on their particular use of datatypes.

  (Contributed by Antoine Pitrou, bpo-4688.)

- Enabling a configure option named `--with-computed-gotos` on compilers that support it (notably: gcc, SunPro, icc), the bytecode evaluation loop is compiled with a new dispatch mechanism which gives speedups of up to 20%, depending on the system, the compiler, and the benchmark.

  (Contributed by Antoine Pitrou along with a number of other participants, bpo-4753).

- The decoding of UTF-8, UTF-16 and LATIN-1 is now two to four times faster.
The `json` module now has a C extension to substantially improve its performance. In addition, the API was modified so that `json` works only with `str`, not with `bytes`. That change makes the module closely match the JSON specification which is defined in terms of Unicode.

Unpickling now interns the attribute names of pickled objects. This saves memory and allows pickles to be smaller.
IDLE

- IDLE’s format menu now provides an option to strip trailing whitespace from a source file.

(Contributed by Roger D. Serwy; bpo-5150.)
Build and C API Changes

Changes to Python’s build process and to the C API include:

- Integers are now stored internally either in base $2^{15}$ or in base $2^{30}$, the base being determined at build time. Previously, they were always stored in base $2^{15}$. Using base $2^{30}$ gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base $2^{30}$ on 64-bit machines and base $2^{15}$ on 32-bit machines; on Unix, there’s a new configure option `--enable-big-digits` that can be used to override this default.

Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there’s a new `sys.int_info` that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

```python
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

(Contributed by Mark Dickinson; [bpo-4258](https://bugs.python.org/issue4258).)

- The `PyLong_AsUnsignedLongLong()` function now handles a negative `pylong` by raising `OverflowError` instead of `TypeError`.

(Contributed by Mark Dickinson and Lisandro Dalcrin; [bpo-5175](https://bugs.python.org/issue5175).)
• Deprecated PyNumber_Int(). Use PyNumber_Long() instead.
  (Contributed by Mark Dickinson; bpo-4910.)

• Added a new PyOS_string_to_double() function to replace the deprecated functions PyOS_ascii_strtod() and PyOS_ascii_atof().
  (Contributed by Mark Dickinson; bpo-5914.)

• Added PyCapsule as a replacement for the PyCObject API. The principal difference is that the new type has a well defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.
  (Contributed by Larry Hastings; bpo-5630.)
Porting to Python 3.1

This section lists previously described changes and other bugfixes that may require changes to your code:

- The new floating point string representations can break existing doctests. For example:

  ```python
def e():
    '''Compute the base of natural logarithms.'''
    return sum(1/math.factorial(x) for x in reversed(doctest.testmod())

  >>> e()
  2.7182818284590451

  Failed example:
  e()
  Expected:
    2.7182818284590451
  Got:
    2.718281828459045
```

- The automatic name remapping in the pickle module for protocol 2 or lower can make Python 3.1 pickles unreadable in Python 3.0. One solution is to use protocol 3. Another solution is to set the `fix_imports` option to `False`. See the discussion above for more details.
What’s New In Python 3.0

Author: Guido van Rossum

This article explains the new features in Python 3.0, compared to 2.6. Python 3.0, also known as “Python 3000” or “Py3K”, is the first ever intentionally backwards incompatible Python release. There are more changes than in a typical release, and more that are important for all Python users. Nevertheless, after digesting the changes, you’ll find that Python really hasn’t changed all that much – by and large, we’re mostly fixing well-known annoyances and warts, and removing a lot of old cruft.

This article doesn’t attempt to provide a complete specification of all new features, but instead tries to give a convenient overview. For full details, you should refer to the documentation for Python 3.0, and/or the many PEPs referenced in the text. If you want to understand the complete implementation and design rationale for a particular feature, PEPs usually have more details than the regular documentation; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented.

Due to time constraints this document is not as complete as it should have been. As always for a new release, the Misc/NEWS file in the source distribution contains a wealth of detailed information about every small thing that was changed.
Common Stumbling Blocks
This section lists those few changes that are most likely to trip you up
if you’re used to Python 2.5.

Print Is A Function
The print statement has been replaced with a print() function,
with keyword arguments to replace most of the special syntax of the
old print statement (PEP 3105). Examples:
Old: print "The answer is", 2*2
New: print("The answer is", 2*2)
Old: print x,
New: print(x, end=" ")

# Trailing comma suppresses newline
# Appends a space instead of a newli

Old: print
New: print()

# Prints a newline
# You must call the function!

Old: print >>sys.stderr, "fatal error"
New: print("fatal error", file=sys.stderr)
Old: print (x, y)
New: print((x, y))

# prints repr((x, y))
# Not the same as print(x, y)!

You can also customize the separator between items, e.g.:
print("There are <", 2**32, "> possibilities!", sep=

which produces:


There are $<4294967296>$ possibilities!

Note:

- The `print()` function doesn't support the "softspace" feature of the old `print` statement. For example, in Python 2.x, `print "A\n", "B"` would write "A\nB\n"; but in Python 3.0, `print("A\n", "B")` writes "A\n B\n".
- Initially, you’ll be finding yourself typing the old `print x` a lot in interactive mode. Time to retrain your fingers to type `print(x)` instead!
- When using the `2to3` source-to-source conversion tool, all `print` statements are automatically converted to `print()` function calls, so this is mostly a non-issue for larger projects.

Views And Iterators Instead Of Lists

Some well-known APIs no longer return lists:

- `dict` methods `dict.keys()`, `dict.items()` and `dict.values()` return "views" instead of lists. For example, this no longer works: `k = d.keys(); k.sort()`. Use `k = sorted(d)` instead (this works in Python 2.5 too and is just as efficient).
- Also, the `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` methods are no longer supported.
- `map()` and `filter()` return iterators. If you really need a list and the input sequences are all of equal length, a quick fix is to wrap `map()` in `list()`, e.g. `list(map(...))`, but a better fix is often to use a list comprehension (especially when the original
code uses `lambda`, or rewriting the code so it doesn't need a list at all. Particularly tricky is `map()` invoked for the side effects of the function; the correct transformation is to use a regular `for` loop (since creating a list would just be wasteful).

If the input sequences are not of equal length, `map()` will stop at the termination of the shortest of the sequences. For full compatibility with `map()` from Python 2.x, also wrap the sequences in `itertools.zip_longest()`, e.g. `map(func, *sequences)` becomes `list(map(func, itertools.zip_longest(*sequences)))`.

- `range()` now behaves like `xrange()` used to behave, except it works with values of arbitrary size. The latter no longer exists.

- `zip()` now returns an iterator.

**Ordering Comparisons**

Python 3.0 has simplified the rules for ordering comparisons:

- The ordering comparison operators (`<`, `<=`, `>=`, `>`) raise a `TypeError` exception when the operands don't have a meaningful natural ordering. Thus, expressions like `1 < ''`, `0 > None` or `len <= len` are no longer valid, and e.g. `None < None` raises `TypeError` instead of returning `False`. A corollary is that sorting a heterogeneous list no longer makes sense – all the elements must be comparable to each other. Note that this does not apply to the `==` and `!=` operators: objects of different incomparable types always compare unequal to each other.

- `builtin.sorted()` and `list.sort()` no longer accept the `cmp` argument providing a comparison function. Use the `key`
argument instead. N.B. the key and reverse arguments are now “keyword-only”.

- The cmp() function should be treated as gone, and the __cmp__() special method is no longer supported. Use __lt__() for sorting, __eq__() with __hash__(), and other rich comparisons as needed. (If you really need the cmp() functionality, you could use the expression (a > b) - (a < b) as the equivalent for cmp(a, b).)

Integers

- **PEP 237**: Essentially, long renamed to int. That is, there is only one built-in integral type, named int; but it behaves mostly like the old long type.
- **PEP 238**: An expression like 1/2 returns a float. Use 1//2 to get the truncating behavior. (The latter syntax has existed for years, at least since Python 2.2.)
- The sys.maxint constant was removed, since there is no longer a limit to the value of integers. However, sys.maxsize can be used as an integer larger than any practical list or string index. It conforms to the implementation’s “natural” integer size and is typically the same as sys.maxint in previous releases on the same platform (assuming the same build options).
- The repr() of a long integer doesn't include the trailing L anymore, so code that unconditionally strips that character will chop off the last digit instead. (Use str() instead.)
- Octal literals are no longer of the form 0720; use 0o720 instead.

Text Vs. Data Instead Of Unicode Vs. 8-bit

Everything you thought you knew about binary data and Unicode has changed.
• Python 3.0 uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. All text is Unicode; however encoded Unicode is represented as binary data. The type used to hold text is `str`, the type used to hold data is `bytes`. The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.0 raises `TypeError`, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but you would get `UnicodeDecodeError` if it contained non-ASCII values. This value-specific behavior has caused numerous sad faces over the years.

• As a consequence of this change in philosophy, pretty much all code that uses Unicode, encodings or binary data most likely has to change. The change is for the better, as in the 2.x world there were numerous bugs having to do with mixing encoded and unencoded text. To be prepared in Python 2.x, start using `unicode` for all unencoded text, and `str` for binary or encoded data only. Then the `2to3` tool will do most of the work for you.

• You can no longer use `u"..."` literals for Unicode text. However, you must use `b"..."` literals for binary data.

• As the `str` and `bytes` types cannot be mixed, you must always explicitly convert between them. Use `str.encode()` to go from `str` to `bytes`, and `bytes.decode()` to go from `bytes` to `str`. You can also use `bytes(s, encoding=...)` and `str(b, encoding=...)`, respectively.

• Like `str`, the `bytes` type is immutable. There is a separate `mutable` type to hold buffered binary data, `bytearray`. Nearly all APIs that accept `bytes` also accept `bytearray`. The mutable API is based on `collections.MutableSequence`.

• All backslashes in raw string literals are interpreted literally. This means that `\U` and `\u` escapes in raw strings are not treated specially. For example, `r'\u20ac'` is a string of 6 characters in Python 3.0, whereas in 2.6, `ur'\u20ac'` was the
single “euro” character. (Of course, this change only affects raw string literals; the euro character is '\u20ac' in Python 3.0.)

- The built-in basestring abstract type was removed. Use `str` instead. The `str` and `bytes` types don’t have functionality enough in common to warrant a shared base class. The 2to3 tool (see below) replaces every occurrence of basestring with `str`.

- Files opened as text files (still the default mode for `open()`) always use an encoding to map between strings (in memory) and bytes (on disk). Binary files (opened with a `b` in the mode argument) always use bytes in memory. This means that if a file is opened using an incorrect mode or encoding, I/O will likely fail loudly, instead of silently producing incorrect data. It also means that even Unix users will have to specify the correct mode (text or binary) when opening a file. There is a platform-dependent default encoding, which on Unixy platforms can be set with the `LANG` environment variable (and sometimes also with some other platform-specific locale-related environment variables). In many cases, but not all, the system default is UTF-8; you should never count on this default. Any application reading or writing more than pure ASCII text should probably have a way to override the encoding. There is no longer any need for using the encoding-aware streams in the `codecs` module.

- The initial values of `sys.stdin`, `sys.stdout` and `sys.stderr` are now unicode-only text files (i.e., they are instances of `io.TextIOBase`). To read and write bytes data with these streams, you need to use their `io.TextIOBase.buffer` attribute.

- Filenames are passed to and returned from APIs as (Unicode) strings. This can present platform-specific problems because on some platforms filenames are arbitrary byte strings. (On the other hand, on Windows filenames are natively stored as Unicode.) As a work-around, most APIs (e.g. `open()`) and many functions in
the `os` module) that take filenames accept `bytes` objects as well as strings, and a few APIs have a way to ask for a `bytes` return value. Thus, `os.listdir()` returns a list of `bytes` instances if the argument is a `bytes` instance, and `os.getcwdb()` returns the current working directory as a `bytes` instance. Note that when `os.listdir()` returns a list of strings, filenames that cannot be decoded properly are omitted rather than raising `UnicodeError`.

- Some system APIs like `os.environ` and `sys.argv` can also present problems when the bytes made available by the system is not interpretable using the default encoding. Setting the `LANG` variable and rerunning the program is probably the best approach.
- **PEP 3138**: The `repr()` of a string no longer escapes non-ASCII characters. It still escapes control characters and code points with non-printable status in the Unicode standard, however.
- **PEP 3120**: The default source encoding is now UTF-8.
- **PEP 3131**: Non-ASCII letters are now allowed in identifiers. (However, the standard library remains ASCII-only with the exception of contributor names in comments.)
- The `StringIO` and `cStringIO` modules are gone. Instead, import the `io` module and use `io.StringIO` or `io.BytesIO` for text and data respectively.
- See also the [Unicode HOWTO](https://docs.python.org/3/howto/unicode.html), which was updated for Python 3.0.
Overview Of Syntax Changes

This section gives a brief overview of every *syntactic* change in Python 3.0.

New Syntax

- **PEP 3107**: Function argument and return value annotations. This provides a standardized way of annotating a function's parameters and return value. There are no semantics attached to such annotations except that they can be introspected at runtime using the `__annotations__` attribute. The intent is to encourage experimentation through metaclasses, decorators or frameworks.

- **PEP 3102**: Keyword-only arguments. Named parameters occurring after `*args` in the parameter list *must* be specified using keyword syntax in the call. You can also use a bare `*` in the parameter list to indicate that you don’t accept a variable-length argument list, but you do have keyword-only arguments.

- Keyword arguments are allowed after the list of base classes in a class definition. This is used by the new convention for specifying a metaclass (see next section), but can be used for other purposes as well, as long as the metaclass supports it.

- **PEP 3104**: `nonlocal` statement. Using `nonlocal x` you can now assign directly to a variable in an outer (but non-global) scope. `nonlocal` is a new reserved word.

- **PEP 3132**: Extended Iterable Unpacking. You can now write
things like \texttt{a, b, *rest = some\_sequence}. And even \texttt{*rest, a = stuff}. The \texttt{rest} object is always a (possibly empty) list; the right-hand side may be any iterable. Example:

\begin{verbatim}
(a, *rest, b) = range(5)
\end{verbatim}

This sets \texttt{a} to 0, \texttt{b} to 4, and \texttt{rest} to [1, 2, 3].

- Dictionary comprehensions: \{k: v for k, v in stuff\} means the same thing as \texttt{dict(stuff)} but is more flexible. (This is \texttt{PEP 274} vindicated. :-)

- Set literals, e.g. \{1, 2\}. Note that {} is an empty dictionary; use \texttt{set()} for an empty set. Set comprehensions are also supported; e.g., \{x for x in stuff\} means the same thing as \texttt{set(stuff)} but is more flexible.

- New octal literals, e.g. \texttt{0o720} (already in 2.6). The old octal literals (0720) are gone.

- New binary literals, e.g. \texttt{0b1010} (already in 2.6), and there is a new corresponding built-in function, \texttt{bin()}.

- Bytes literals are introduced with a leading \texttt{b} or \texttt{B}, and there is a new corresponding built-in function, \texttt{bytes()}.

**Changed Syntax**

- \texttt{PEP 3109} and \texttt{PEP 3134}: new \texttt{raise} statement syntax: \texttt{raise [expr [from expr]]}. See below.

- \texttt{as} and \texttt{with} are now reserved words. (Since 2.6, actually.)
- True, False, and None are reserved words. (2.6 partially enforced the restrictions on None already.)

- Change from except exc, var to except exc as var. See PEP 3110.

- PEP 3115: New Metaclass Syntax. Instead of:

  ```python
  class C:
      __metaclass__ = M
  ...
  ```

  you must now use:

  ```python
  class C(metaclass=M):
      ...
  ```

  The module-global __metaclass__ variable is no longer supported. (It was a crutch to make it easier to default to new-style classes without deriving every class from object.)

- List comprehensions no longer support the syntactic form `[... for var in item1, item2, ...]. Use `[... for var in (item1, item2, ...)]]` instead. Also note that list comprehensions have different semantics: they are closer to syntactic sugar for a generator expression inside a list() constructor, and in particular the loop control variables are no longer leaked into the surrounding scope.

- The ellipsis (...) can be used as an atomic expression anywhere. (Previously it was only allowed in slices.) Also, it must now be spelled as . . . . (Previously it could also be spelled as . . . , by a mere accident of the grammar.)
PEP 3113: Tuple parameter unpacking removed. You can no longer write `def foo(a, (b, c)): ....` Use `def foo(a, b_c): b, c = b_c` instead.

- Removed backticks (use `repr()` instead).
- Removed `<>` (use `!=` instead).
- Removed keyword: `exec()` is no longer a keyword; it remains as a function. (Fortunately the function syntax was also accepted in 2.x.) Also note that `exec()` no longer takes a stream argument; instead of `exec(f)` you can use `exec(f.read())`.
- Integer literals no longer support a trailing `l` or `L`.
- String literals no longer support a leading `u` or `U`.
- The `from module import *` syntax is only allowed at the module level, no longer inside functions.
- The only acceptable syntax for relative imports is `from [module] import name`. All `import` forms not starting with `.` are interpreted as absolute imports. (PEP 328)
- Classic classes are gone.
Changes Already Present In Python 2.6

Since many users presumably make the jump straight from Python 2.5 to Python 3.0, this section reminds the reader of new features that were originally designed for Python 3.0 but that were back-ported to Python 2.6. The corresponding sections in What's New in Python 2.6 should be consulted for longer descriptions.

- **PEP 343: The ‘with’ statement.** The `with` statement is now a standard feature and no longer needs to be imported from the `__future__`. Also check out Writing Context Managers and The contextlib module.
- **PEP 366: Explicit Relative Imports From a Main Module.** This enhances the usefulness of the `-m` option when the referenced module lives in a package.
- **PEP 370: Per-user site-packages Directory.**
- **PEP 371: The multiprocessing Package.**
- **PEP 3101: Advanced String Formatting.** Note: the 2.6 description mentions the `format()` method for both 8-bit and Unicode strings. In 3.0, only the `str` type (text strings with Unicode support) supports this method; the `bytes` type does not. The plan is to eventually make this the only API for string formatting, and to start deprecating the `%` operator in Python 3.1.
- **PEP 3105: print As a Function.** This is now a standard feature and no longer needs to be imported from `__future__`. More details were given above.
- **PEP 3110: Exception-Handling Changes.** The `except exc as var` syntax is now standard and except `exc`, `var` is no longer supported. (Of course, the `as var` part is still optional.)
- **PEP 3112: Byte Literals.** The `b"..."` string literal notation (and its variants like `b'...', b""""..."""", and br"..."`) now produces a literal of type `bytes`. 
- **PEP 3116: New I/O Library.** The `io` module is now the standard way of doing file I/O. The built-in `open()` function is now an alias for `io.open()` and has additional keyword arguments `encoding`, `errors`, `newline` and `closefd`. Also note that an invalid `mode` argument now raises `ValueError`, not `IOError`. The binary file object underlying a text file object can be accessed as `f.buffer` (but beware that the text object maintains a buffer of itself in order to speed up the encoding and decoding operations).

- **PEP 3118: Revised Buffer Protocol.** The old built-in `buffer()` is now really gone; the new built-in `memoryview()` provides (mostly) similar functionality.

- **PEP 3119: Abstract Base Classes.** The `abc` module and the ABCs defined in the `collections` module plays a somewhat more prominent role in the language now, and built-in collection types like `dict` and `list` conform to the `collections.MutableMapping` and `collections.MutableSequence` ABCs, respectively.

- **PEP 3127: Integer Literal Support and Syntax.** As mentioned above, the new octal literal notation is the only one supported, and binary literals have been added.

- **PEP 3129: Class Decorators.**

- **PEP 3141: A Type Hierarchy for Numbers.** The `numbers` module is another new use of ABCs, defining Python’s “numeric tower”. Also note the new `fractions` module which implements `numbers.Rational`. 
Library Changes

Due to time constraints, this document does not exhaustively cover the very extensive changes to the standard library. **PEP 3108** is the reference for the major changes to the library. Here's a capsule review:

- Many old modules were removed. Some, like gopherlib (no longer used) and md5 (replaced by **hashlib**), were already deprecated by **PEP 4**. Others were removed as a result of the removal of support for various platforms such as Irix, BeOS and Mac OS 9 (see **PEP 11**). Some modules were also selected for removal in Python 3.0 due to lack of use or because a better replacement exists. See **PEP 3108** for an exhaustive list.

- The bsddb3 package was removed because its presence in the core standard library has proved over time to be a particular burden for the core developers due to testing instability and Berkeley DB’s release schedule. However, the package is alive and well, externally maintained at [https://www.jcea.es/programacion/pybsddb.htm](https://www.jcea.es/programacion/pybsddb.htm).

- Some modules were renamed because their old name disobeyed **PEP 8**, or for various other reasons. Here’s the list:

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>_winreg</td>
<td>winreg</td>
</tr>
<tr>
<td>ConfigParser</td>
<td>configparser</td>
</tr>
<tr>
<td>copy_reg</td>
<td>copyreg</td>
</tr>
<tr>
<td>Queue</td>
<td>queue</td>
</tr>
<tr>
<td>SocketServer</td>
<td>socketserver</td>
</tr>
</tbody>
</table>
A common pattern in Python 2.x is to have one version of a module implemented in pure Python, with an optional accelerated version implemented as a C extension; for example, pickle and cPickle. This places the burden of importing the accelerated version and falling back on the pure Python version on each user of these modules. In Python 3.0, the accelerated versions are considered implementation details of the pure Python versions. Users should always import the standard version, which attempts to import the accelerated version and falls back to the pure Python version. The pickle / cPickle pair received this treatment. The profile module is on the list for 3.1. The StringIO module has been turned into a class in the io module.

Some related modules have been grouped into packages, and usually the submodule names have been simplified. The resulting new packages are:

- **dbm** (anydbm, dbhash, dbm, dumbdbm, gdbm, whichdb).
- **html** (HTMLParser, htmlentitydefs).
- **http** (httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib).
- **tkinter** (all Tkinter-related modules except turtle). The target audience of turtle doesn’t really care about tkinter. Also note that as of Python 2.6, the functionality of turtle has been greatly enhanced.
- **urllib** (urllib, urllib2, urlparse, robotparser).
- **xmlrpc** (xmlrpclib, DocXMLRPCServer,
Some other changes to standard library modules, not covered by PEP 3108:

- Killed sets. Use the built-in `set()` class.
- Cleanup of the `sys` module: removed `sys.exitfunc()`, `sys.exc_clear()`, `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`. (Note that `sys.last_type` etc. remain.)
- Cleanup of the `array.array` type: the `read()` and `write()` methods are gone; use `fromfile()` and `tofile()` instead. Also, the `c` typecode for array is gone – use either `b` for bytes or `u` for Unicode characters.
- Cleanup of the `operator` module: removed `sequenceIncludes()` and `iscallable()`.
- Cleanup of the `thread` module: `acquire_lock()` and `release_lock()` are gone; use `acquire()` and `release()` instead.
- Cleanup of the `random` module: removed the `jumpahead()` API.
- The new module is gone.
- The functions `os.tmpnam()`, `os.tempnam()` and `os.tmpfile()` have been removed in favor of the `tempfile` module.
- The `tokenizer` module has been changed to work with bytes. The main entry point is now `tokenizer.tokenize()`, instead of `generate_tokens`.
- `string.letters` and its friends (`string.lowercase` and `string.uppercase`) are gone. Use `string.ascii_letters` etc. instead. (The reason for the removal is that `string.letters` and friends had locale-specific behavior, which is a bad idea for such attractively-named global “constants”.)
- Renamed module `__builtin__` to `builtins` (removing the
underscores, adding an ‘s’). The __builtins__ variable found in most global namespaces is unchanged. To modify a builtin, you should use builtins, not __builtins__!
**PEP 3101**: A New Approach To String Formatting

- A new system for built-in string formatting operations replaces the % string formatting operator. (However, the % operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.) Read **PEP 3101** for the full scoop.
Changes To Exceptions

The APIs for raising and catching exception have been cleaned up and new powerful features added:

- **PEP 352**: All exceptions must be derived (directly or indirectly) from `BaseException`. This is the root of the exception hierarchy. This is not new as a recommendation, but the *requirement* to inherit from `BaseException` is new. (Python 2.6 still allowed classic classes to be raised, and placed no restriction on what you can catch.) As a consequence, string exceptions are finally truly and utterly dead.

- Almost all exceptions should actually derive from `Exception`; `BaseException` should only be used as a base class for exceptions that should only be handled at the top level, such as `SystemExit` or `KeyboardInterrupt`. The recommended idiom for handling all exceptions except for this latter category is to use `except Exception`.

- `StandardError` was removed.

- Exceptions no longer behave as sequences. Use the `args` attribute instead.

- **PEP 3109**: Raising exceptions. You must now use `raise Exception(args)` instead of `raise Exception, args`. Additionally, you can no longer explicitly specify a traceback; instead, if you *have* to do this, you can assign directly to the `__traceback__` attribute (see below).

- **PEP 3110**: Catching exceptions. You must now use `except`
SomeException as variable instead of except SomeException, variable. Moreover, the variable is explicitly deleted when the except block is left.

- **PEP 3134**: Exception chaining. There are two cases: implicit chaining and explicit chaining. Implicit chaining happens when an exception is raised in an except or finally handler block. This usually happens due to a bug in the handler block; we call this a secondary exception. In this case, the original exception (that was being handled) is saved as the __context__ attribute of the secondary exception. Explicit chaining is invoked with this syntax:

```python
raise SecondaryException() from primary_exception
```

(where primary_exception is any expression that produces an exception object, probably an exception that was previously caught). In this case, the primary exception is stored on the __cause__ attribute of the secondary exception. The traceback printed when an unhandled exception occurs walks the chain of __cause__ and __context__ attributes and prints a separate traceback for each component of the chain, with the primary exception at the top. (Java users may recognize this behavior.)

- **PEP 3134**: Exception objects now store their traceback as the __traceback__ attribute. This means that an exception object now contains all the information pertaining to an exception, and there are fewer reasons to use `sys.exc_info()` (though the latter is not removed).

- A few exception messages are improved when Windows fails to load an extension module. For example, error code 193 is
now %1 is not a valid Win32 application. Strings now deal with non-English locales.
Miscellaneous Other Changes

Operators And Special Methods

- `!=` now returns the opposite of `==`, unless `==` returns `NotImplemented`.
- The concept of “unbound methods” has been removed from the language. When referencing a method as a class attribute, you now get a plain function object.
- `__getslice__()`, `__setslice__()` and `__delslice__()` were killed. The syntax `a[i:j]` now translates to `a.__getitem__(slice(i, j))` (or `__setitem__()` or `__delitem__()`), when used as an assignment or deletion target, respectively.
- **PEP 3114**: the standard `next()` method has been renamed to `__next__()`. 
- The `__oct__()` and `__hex__()` special methods are removed – `oct()` and `hex()` use `__index__()` now to convert the argument to an integer.
- Removed support for `__members__` and `__methods__`.
- The function attributes named `func_X` have been renamed to use the `__X__` form, freeing up these names in the function attribute namespace for user-defined attributes. To wit, `func_closure`, `func_code`, `func_defaults`, `func_dict`, `func_doc`, `func_globals`, `func_name` were renamed to `__closure__`, `__code__`, `__defaults__`, `__dict__`, `__doc__`, `__globals__`, `__name__`, respectively.
- `__nonzero__()` is now `__bool__()`.

Builtins
• **PEP 3135**: New super(). You can now invoke super() without arguments and (assuming this is in a regular instance method defined inside a class statement) the right class and instance will automatically be chosen. With arguments, the behavior of super() is unchanged.

• **PEP 3111**: raw_input() was renamed to input(). That is, the new input() function reads a line from sys.stdin and returns it with the trailing newline stripped. It raises EOFError if the input is terminated prematurely. To get the old behavior of input(), use eval(input()).

A new built-in function next() was added to call the __next__() method on an object.

The round() function rounding strategy and return type have changed. Exact halfway cases are now rounded to the nearest even result instead of away from zero. (For example, round(2.5) now returns 2 rather than 3.) round(x[, n]) now delegates to x.__round__(n) instead of always returning a float. It generally returns an integer when called with a single argument and a value of the same type as x when called with two arguments.

• Moved intern() to sys.intern().

• Removed: apply(). Instead of apply(f, args) use f(*args).

• Removed callable(). Instead of callable(f) you can use isinstance(f, collections.Callable). The operator.isCallable() function is also gone.

• Removed coerce(). This function no longer serves a purpose now that classic classes are gone.

• Removed execfile(). Instead of execfile(fn) use exec(open(fn).read()).

• Removed the file type. Use open(). There are now several different kinds of streams that open can return in the io module.
• Removed `reduce()`. Use `functools.reduce()` if you really need it; however, 99 percent of the time an explicit `for` loop is more readable.
• Removed `reload()`. Use `imp.reload()`.
• Removed `dict.has_key()` – use the `in` operator instead.
Build and C API Changes

Due to time constraints, here is a very incomplete list of changes to the C API.

- Support for several platforms was dropped, including but not limited to Mac OS 9, BeOS, RISCOS, Irix, and Tru64.
- **PEP 3118**: New Buffer API.
- **PEP 3121**: Extension Module Initialization & Finalization.
- **PEP 3123**: Making `PyObject_HEAD` conform to standard C.
- No more C API support for restricted execution.
- `PyNumber_Coerce()`, `PyNumber_CoerceEx()`, `PyMember_Get()`, and `PyMember_Set()` C APIs are removed.
- New C API `PyImport_ImportModuleNoBlock()`, works like `PyImport_ImportModule()` but won't block on the import lock (returning an error instead).
- Renamed the boolean conversion C-level slot and method: `nb_nonzero` is now `nb_bool`.
- Removed `METH_OLDARGS` and `WITH_CYCLE_GC` from the C API.
Performance

The net result of the 3.0 generalizations is that Python 3.0 runs the pystone benchmark around 10% slower than Python 2.5. Most likely the biggest cause is the removal of special-casing for small integers. There's room for improvement, but it will happen after 3.0 is released!
Porting To Python 3.0

For porting existing Python 2.5 or 2.6 source code to Python 3.0, the best strategy is the following:

0. (Prerequisite:) Start with excellent test coverage.
1. Port to Python 2.6. This should be no more work than the average port from Python 2.x to Python 2.(x+1). Make sure all your tests pass.
2. (Still using 2.6:) Turn on the -3 command line switch. This enables warnings about features that will be removed (or change) in 3.0. Run your test suite again, and fix code that you get warnings about until there are no warnings left, and all your tests still pass.
3. Run the 2to3 source-to-source translator over your source code tree. (See 2to3 - Automated Python 2 to 3 code translation for more on this tool.) Run the result of the translation under Python 3.0. Manually fix up any remaining issues, fixing problems until all tests pass again.

It is not recommended to try to write source code that runs unchanged under both Python 2.6 and 3.0; you’d have to use a very contorted coding style, e.g. avoiding print statements, metaclasses, and much more. If you are maintaining a library that needs to support both Python 2.6 and Python 3.0, the best approach is to modify step 3 above by editing the 2.6 version of the source code and running the 2to3 translator again, rather than editing the 3.0 version of the source code.

For porting C extensions to Python 3.0, please see Porting Extension Modules to Python 3.
What’s New in Python 2.7

Author: A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.7. Python 2.7 was released on July 3, 2010.

Numeric handling has been improved in many ways, for both floating-point numbers and for the Decimal class. There are some useful additions to the standard library, such as a greatly enhanced unittest module, the argparse module for parsing command-line options, convenient OrderedDict and Counter classes in the collections module, and many other improvements.

Python 2.7 is planned to be the last of the 2.x releases, so we worked on making it a good release for the long term. To help with porting to Python 3, several new features from the Python 3.x series have been included in 2.7.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.7 at https://docs.python.org. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature or the issue on https://bugs.python.org in which a change was discussed. Whenever possible, “What’s New in Python” links to the bug/patch item for each change.
The Future for Python 2.x

Python 2.7 is the last major release in the 2.x series, as the Python maintainers have shifted the focus of their new feature development efforts to the Python 3.x series. This means that while Python 2 continues to receive bug fixes, and to be updated to build correctly on new hardware and versions of supported operated systems, there will be no new full feature releases for the language or standard library.

However, while there is a large common subset between Python 2.7 and Python 3, and many of the changes involved in migrating to that common subset, or directly to Python 3, can be safely automated, some other changes (notably those associated with Unicode handling) may require careful consideration, and preferably robust automated regression test suites, to migrate effectively.

This means that Python 2.7 will remain in place for a long time, providing a stable and supported base platform for production systems that have not yet been ported to Python 3. The full expected lifecycle of the Python 2.7 series is detailed in PEP 373.

Some key consequences of the long-term significance of 2.7 are:

- As noted above, the 2.7 release has a much longer period of maintenance when compared to earlier 2.x versions. Python 2.7 is currently expected to remain supported by the core development team (receiving security updates and other bug fixes) until at least 2020 (10 years after its initial release, compared to the more typical support period of 18–24 months).
- As the Python 2.7 standard library ages, making effective use of the Python Package Index (either directly or via a redistributor) becomes more important for Python 2 users. In addition to a wide variety of third party packages for various tasks, the available
packages include backports of new modules and features from the Python 3 standard library that are compatible with Python 2, as well as various tools and libraries that can make it easier to migrate to Python 3. The Python Packaging User Guide provides guidance on downloading and installing software from the Python Package Index.

- While the preferred approach to enhancing Python 2 is now the publication of new packages on the Python Package Index, this approach doesn’t necessarily work in all cases, especially those related to network security. In exceptional cases that cannot be handled adequately by publishing new or updated packages on PyPI, the Python Enhancement Proposal process may be used to make the case for adding new features directly to the Python 2 standard library. Any such additions, and the maintenance releases where they were added, will be noted in the New Features Added to Python 2.7 Maintenance Releases section below.

For projects wishing to migrate from Python 2 to Python 3, or for library and framework developers wishing to support users on both Python 2 and Python 3, there are a variety of tools and guides available to help decide on a suitable approach and manage some of the technical details involved. The recommended starting point is the Porting Python 2 Code to Python 3 HOWTO guide.
Changes to the Handling of Deprecation Warnings

For Python 2.7, a policy decision was made to silence warnings only of interest to developers by default. `DeprecationWarning` and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that became Python 3.2. (Discussed on stdlib-sig and carried out in bpo-7319.)

In previous releases, `DeprecationWarning` messages were enabled by default, providing Python developers with a clear indication of where their code may break in a future major version of Python.

However, there are increasingly many users of Python-based applications who are not directly involved in the development of those applications. `DeprecationWarning` messages are irrelevant to such users, making them worry about an application that's actually working correctly and burdening application developers with responding to these concerns.

You can re-enable display of `DeprecationWarning` messages by running Python with the `-Wdefault` (short form: `-Wd`) switch, or by setting the `PYTHONWARNINGS` environment variable to "default" (or "d") before running Python. Python code can also re-enable them by calling `warnings.simplefilter('default')`.

The `unittest` module also automatically reenables deprecation warnings when running tests.
Python 3.1 Features

Much as Python 2.6 incorporated features from Python 3.0, version 2.7 incorporates some of the new features in Python 3.1. The 2.x series continues to provide tools for migrating to the 3.x series.

A partial list of 3.1 features that were backported to 2.7:

- The syntax for set literals ({1, 2, 3} is a mutable set).
- Dictionary and set comprehensions ({i: i*2 for i in range(3)}).
- Multiple context managers in a single `with` statement.
- A new version of the `io` library, rewritten in C for performance.
- The ordered-dictionary type described in PEP 372: Adding an Ordered Dictionary to collections.
- The new ""," format specifier described in PEP 378: Format Specifier for Thousands Separator.
- The `memoryview` object.
- A small subset of the `importlib` module, described below.
- The `repr()` of a float \( x \) is shorter in many cases: it's now based on the shortest decimal string that's guaranteed to round back to \( x \). As in previous versions of Python, it's guaranteed that `float(repr(x))` recovers \( x \).
- Float-to-string and string-to-float conversions are correctly rounded. The `round()` function is also now correctly rounded.
- The `PyCapsule` type, used to provide a C API for extension modules.
- The `PyLong_AsLongAndOverflow()` C API function.

Other new Python3-mode warnings include:

- `operator.isCallable()` and
operator.sequenceIncludes(), which are not supported in 3.x, now trigger warnings.

- The -3 switch now automatically enables the -Qwarn switch that causes warnings about using classic division with integers and long integers.
PEP 372: Adding an Ordered Dictionary to collections

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, 2.7 introduces a new `OrderedDict` class in the `collections` module.

The `OrderedDict` API provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```python
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1), ...
...     ('second', 2), ...
...     ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```python
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```python
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```
The `popitem()` method has an optional `last` argument that defaults to `True`. If `last` is true, the most recently added key is returned and removed; if it's false, the oldest key is selected:

```python
>>> od = OrderedDict([(x, 0) for x in range(20)])
>>> od.popitem()
(19, 0)
>>> od.popitem()
(18, 0)
>>> od.popitem(last=False)
(0, 0)
>>> od.popitem(last=False)
(1, 0)
```

Comparing two ordered dictionaries checks both the keys and values, and requires that the insertion order was the same:

```python
>>> od1 = OrderedDict([('first', 1), ...
                        ('second', 2), ...
                        ('third', 3)])
>>> od2 = OrderedDict([('third', 3), ...
                        ('first', 1), ...
                        ('second', 2)])
>>> od1 == od2
False
>>> # Move 'third' key to the end
>>> del od2['third']; od2['third'] = 3
>>> od1 == od2
True
```

Comparing an `OrderedDict` with a regular dictionary ignores the insertion order and just compares the keys and values.

How does the `OrderedDict` work? It maintains a doubly-linked list of keys, appending new keys to the list as they're inserted. A secondary
dictionary maps keys to their corresponding list node, so deletion doesn't have to traverse the entire linked list and therefore remains $O(1)$.

The standard library now supports use of ordered dictionaries in several modules.

- The ConfigParser module uses them by default, meaning that configuration files can now be read, modified, and then written back in their original order.
- The `asdict()` method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices.
- The `json` module's `JSONDecoder` class constructor was extended with an `object_pairs_hook` parameter to allow `OrderedDict` instances to be built by the decoder. Support was also added for third-party tools like PyYAML.

See also:

**PEP 372 - Adding an ordered dictionary to collections**

PEP written by Armin Ronacher and Raymond Hettinger; implemented by Raymond Hettinger.
To make program output more readable, it can be useful to add separators to large numbers, rendering them as 18,446,744,073,709,551,616 instead of 18446744073709551616.

The fully general solution for doing this is the `locale` module, which can use different separators (""," in North America, "." in Europe) and different grouping sizes, but `locale` is complicated to use and unsuitable for multi-threaded applications where different threads are producing output for different locales.

Therefore, a simple comma-grouping mechanism has been added to the mini-language used by the `str.format()` method. When formatting a floating-point number, simply include a comma between the width and the precision:

```python
>>> '{:20,.2f}'.format(18446744073709551616.0)
'18,446,744,073,709,551,616.00'
```

When formatting an integer, include the comma after the width:

```python
>>> '{:20,d}'.format(18446744073709551616)
'18,446,744,073,709,551,616'
```

This mechanism is not adaptable at all; commas are always used as the separator and the grouping is always into three-digit groups. The comma-formatting mechanism isn't as general as the `locale` module, but it's easier to use.

See also:
PEP 378 - Format Specifier for Thousands Separator
PEP written by Raymond Hettinger; implemented by Eric Smith.
PEP 389: The argparse Module for Parsing Command Lines

The `argparse` module for parsing command-line arguments was added as a more powerful replacement for the `optparse` module.

This means Python now supports three different modules for parsing command-line arguments: `getopt`, `optparse`, and `argparse`. The `getopt` module closely resembles the C library's `getopt()` function, so it remains useful if you're writing a Python prototype that will eventually be rewritten in C. `optparse` becomes redundant, but there are no plans to remove it because there are many scripts still using it, and there's no automated way to update these scripts. (Making the `argparse` API consistent with `optparse`'s interface was discussed but rejected as too messy and difficult.)

In short, if you're writing a new script and don't need to worry about compatibility with earlier versions of Python, use `argparse` instead of `optparse`.

Here's an example:

```python
import argparse

parser = argparse.ArgumentParser(description='Command-line example.'

# Add optional switches
parser.add_argument('-v', action='store_true', dest='verbose',
                    help='produce verbose output')
parser.add_argument('-o', action='store', dest='output',
                    metavar='FILE',
                    help='direct output to FILE instead of stdout')
```
```
parser.add_argument('-C', action='store', type=int, 
    metavar='NUM', default=0, 
    help='display NUM lines of added context')

# Allow any number of additional arguments.
parser.add_argument(nargs='*', action='store', dest='inputs', 
    help='input filenames (default is stdin)')

args = parser.parse_args()
print(args.__dict__)
```

Unless you override it, -h and --help switches are automatically added, and produce neatly formatted output:

```
-> ./python.exe argparse-example.py --help
usage: argparse-example.py [-h] [-v] [-o FILE] [-C NUM]

Command-line example.

positional arguments:
  inputs     input filenames (default is stdin)

optional arguments:
  -h, --help  show this help message and exit
  -v          produce verbose output
  -o FILE     direct output to FILE instead of stdout
  -C NUM      display NUM lines of added context
```

As with optparse, the command-line switches and arguments are returned as an object with attributes named by the dest parameters:

```
-> ./python.exe argparse-example.py -v
{'output': None, 
 'is_verbos': True, 
 'context': 0, 
```

'inputs': []

-> ./python.exe argparse-example.py -v -o /tmp/output

{'output': '/tmp/output',
 'is_verbos': True,
 'context': 4,
 'inputs': ['file1', 'file2']}

argparse has much fancier validation than optparse; you can specify an exact number of arguments as an integer, 0 or more arguments by passing '*'. 1 or more by passing '+', or an optional argument with '?'. A top-level parser can contain sub-parsers to define subcommands that have different sets of switches, as in `svn commit`, `svn checkout`, etc. You can specify an argument's type as `FileType`, which will automatically open files for you and understands that '-' means standard input or output.

See also:

argparse documentation
The documentation page of the argparse module.

Upgrading optparse code
Part of the Python documentation, describing how to convert code that uses optparse.

PEP 389 - argparse - New Command Line Parsing Module
PEP written and implemented by Steven Bethard.
PEP 391: Dictionary-Based Configuration For Logging

The `logging` module is very flexible; applications can define a tree of logging subsystems, and each logger in this tree can filter out certain messages, format them differently, and direct messages to a varying number of handlers.

All this flexibility can require a lot of configuration. You can write Python statements to create objects and set their properties, but a complex set-up requires verbose but boring code. `logging` also supports a `fileConfig()` function that parses a file, but the file format doesn't support configuring filters, and it's messier to generate programmatically.

Python 2.7 adds a `dictConfig()` function that uses a dictionary to configure logging. There are many ways to produce a dictionary from different sources: construct one with code; parse a file containing JSON; or use a YAML parsing library if one is installed. For more information see Configuration functions.

The following example configures two loggers, the root logger and a logger named “network”. Messages sent to the root logger will be sent to the system log using the syslog protocol, and messages to the “network” logger will be written to a `network.log` file that will be rotated once the log reaches 1MB.

```python
import logging
import logging.config

configdict = {
    'version': 1,  # Configuration schema in use; must
```
'formatters': {'standard': {'format': ('%(asctime)s (%(name)-15s ' '
%(levelname)-8s %(message)s')}}}}

'handslers': {'netlog': {'backupCount': 10, 'class': 'logging.handlers.RotatingFileHandler', 'filename': '/logs/network.log', 'formatter': 'standard', 'level': 'INFO', 'maxBytes': 1000000}, 'syslog': {'class': 'logging.handlers.SysLogHandler', 'formatter': 'standard', 'level': 'ERROR'}}},

# Specify all the subordinate loggers
'loggers': {'network': {'handlers': ['netlog']}}},

# Specify properties of the root logger
'root': {'handlers': ['syslog']},

# Set up configuration
logging.config.dictConfig(configdict)

# As an example, log two error messages
logger = logging.getLogger('/
logger.error('Database not found')

netlogger = logging.getLogger('network')
netlogger.error('Connection failed')
Three smaller enhancements to the `logging` module, all implemented by Vinay Sajip, are:

- The `SysLogHandler` class now supports syslogging over TCP. The constructor has a `socktype` parameter giving the type of socket to use, either `socket.SOCK_DGRAM` for UDP or `socket.SOCK_STREAM` for TCP. The default protocol remains UDP.
- `Logger` instances gained a `getChild()` method that retrieves a descendant logger using a relative path. For example, once you retrieve a logger by doing `log = getLogger('app')`, calling `log.getChild('network.listen')` is equivalent to `getLogger('app.network.listen')`.
- The `LoggerAdapter` class gained an `isEnabledFor()` method that takes a `level` and returns whether the underlying logger would process a message of that level of importance.

See also:

**PEP 391 - Dictionary-Based Configuration For Logging**
PEP written and implemented by Vinay Sajip.
PEP 3106: Dictionary Views

The dictionary methods `keys()`, `values()`, and `items()` are different in Python 3.x. They return an object called a view instead of a fully materialized list.

It's not possible to change the return values of `keys()`, `values()`, and `items()` in Python 2.7 because too much code would break. Instead the 3.x versions were added under the new names `viewkeys()`, `viewvalues()`, and `viewitems()`.

```python
>>> d = dict((i*10, chr(65+i)) for i in range(26))
>>> d
>>> d.viewkeys()
dict_keys([0, 130, 10, 140, 20, 150, 30, ..., 250])
```

Views can be iterated over, but the key and item views also behave like sets. The `&` operator performs intersection, and `|` performs a union:

```python
>>> d1 = dict((i*10, chr(65+i)) for i in range(26))
>>> d2 = dict((i**.5, i) for i in range(1000))
>>> d1.viewkeys() & d2.viewkeys()
set([0.0, 10.0, 20.0, 30.0])
>>> d1.viewkeys() | range(0, 30)
set([0, 1, 130, 3, 4, 5, 6, ..., 120, 250])
```

The view keeps track of the dictionary and its contents change as the dictionary is modified:

```python
>>> vk = d.viewkeys()
```
However, note that you can't add or remove keys while you're iterating over the view:

```python
>>> for k in vk:
...     d[k*2] = k
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

You can use the view methods in Python 2.x code, and the 2to3 converter will change them to the standard `keys()`, `values()`, and `items()` methods.

See also:

**PEP 3106 - Revamping dict.keys(), .values() and .items()**

PEP written by Guido van Rossum. Backported to 2.7 by Alexandre Vassalotti; bpo-1967.
PEP 3137: The memoryview Object

The `memoryview` object provides a view of another object’s memory content that matches the `bytes` type’s interface.

```python
>>> import string
>>> m = memoryview(string.letters)
>>> m
<memory at 0x37f850>
>>> len(m)  # Returns length of underlying object
52
>>> m[0], m[25], m[26]  # Indexing returns one byte ('a', 'z', 'A')
>>> m2 = m[0:26]  # Slicing returns another memoryview
>>> m2
<memory at 0x37f080>
```

The content of the view can be converted to a string of bytes or a list of integers:

```python
>>> m2.tobytes()
'abcdefgijklmnopqrstuvwxyz'
>>> m2.tolist()
[97, 98, 99, 100, 101, 102, 103, ... 121, 122]
```

`memoryview` objects allow modifying the underlying object if it’s a mutable object.

```python
>>> m2[0] = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
```
>>> b = bytearray(string.letters)  # Creating a mutable object
>>> b
bytearray(b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
>>> mb = memoryview(b)
>>> mb[0] = '*'  # Assign to view, changing the original buffer.
>>> b[0:5]       # The bytearray has been changed.
bytearray(b'*bcde')

See also:

**PEP 3137 - Immutable Bytes and Mutable Buffer**

PEP written by Guido van Rossum. Implemented by Travis Oliphant, Antoine Pitrou and others. Backported to 2.7 by Antoine Pitrou; bpo-2396.
Other Language Changes

Some smaller changes made to the core Python language are:

- The syntax for set literals has been backported from Python 3.x. Curly brackets are used to surround the contents of the resulting mutable set; set literals are distinguished from dictionaries by not containing colons and values. `{}` continues to represent an empty dictionary; use `set()` for an empty set.

```python
>>> {1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
set([1, 2, 3, 4, 5])
>>> set()  # empty set
set([])
>>> {}  # empty dict
{}
```

Backported by Alexandre Vassalotti; bpo-2335.

- Dictionary and set comprehensions are another feature backported from 3.x, generalizing list/generator comprehensions to use the literal syntax for sets and dictionaries.

```python
>>> {x: x*x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> {('a'*x) for x in range(6)}
set(['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa'])
```

Backported by Alexandre Vassalotti; bpo-2333.

- The `with` statement can now use multiple context managers in one statement. Context managers are processed from left to right and each one is treated as beginning a new `with` statement. This
means that:

```python
with A() as a, B() as b:
    ... suite of statements ...
```

is equivalent to:

```python
with A() as a:
    with B() as b:
        ... suite of statements ...
```

The `contextlib.nested()` function provides a very similar function, so it's no longer necessary and has been deprecated.

(Proposed in https://codereview.appspot.com/53094; implemented by Georg Brandl.)

- Conversions between floating-point numbers and strings are now correctly rounded on most platforms. These conversions occur in many different places: `str()` on floats and complex numbers; the `float` and `complex` constructors; numeric formatting; serializing and deserializing floats and complex numbers using the `marshal`, `pickle` and `json` modules; parsing of float and imaginary literals in Python code; and `Decimal`-to-float conversion.

Related to this, the `repr()` of a floating-point number `x` now returns a result based on the shortest decimal string that's guaranteed to round back to `x` under correct rounding (with round-half-to-even rounding mode). Previously it gave a string based on rounding `x` to 17 decimal digits.

The rounding library responsible for this improvement works on
Windows and on Unix platforms using the gcc, icc, or suncc compilers. There may be a small number of platforms where correct operation of this code cannot be guaranteed, so the code is not used on such systems. You can find out which code is being used by checking `sys.float_repr_style`, which will be **short** if the new code is in use and **legacy** if it isn’t.

Implemented by Eric Smith and Mark Dickinson, using David Gay’s `dtoa.c` library; **bpo-7117**.

- Conversions from long integers and regular integers to floating point now round differently, returning the floating-point number closest to the number. This doesn’t matter for small integers that can be converted exactly, but for large numbers that will unavoidably lose precision, Python 2.7 now approximates more closely. For example, Python 2.6 computed the following:

```
>>> n = 295147905179352891391
>>> float(n)
2.9514790517935283e+20
>>> n - long(float(n))
65535L
```

Python 2.7’s floating-point result is larger, but much closer to the true value:

```
>>> n = 295147905179352891391
>>> float(n)
2.9514790517935289e+20
>>> n - long(float(n))
-1L
```

(Implemented by Mark Dickinson; **bpo-3166**.)
Integer division is also more accurate in its rounding behaviours. (Also implemented by Mark Dickinson; bpo-1811.)

- Implicit coercion for complex numbers has been removed; the interpreter will no longer ever attempt to call a `__coerce__()` method on complex objects. (Removed by Meador Inge and Mark Dickinson; bpo-5211.)

- The `str.format()` method now supports automatic numbering of the replacement fields. This makes using `str.format()` more closely resemble using `%s` formatting:

  ```
  >>> '{:}{:}{:}'.format(2009, 04, 'Sunday')
  '2009:4:Sunday'
  >>> '{:}{:}{day}'.format(2009, 4, day='Sunday')
  '2009:4:Sunday'
  ```

  The auto-numbering takes the fields from left to right, so the first `{...}` specifier will use the first argument to `str.format()`, the next specifier will use the next argument, and so on. You can't mix auto-numbering and explicit numbering – either number all of your specifier fields or none of them – but you can mix auto-numbering and named fields, as in the second example above. (Contributed by Eric Smith; bpo-5237.)

  Complex numbers now correctly support usage with `format()`, and default to being right-aligned. Specifying a precision or comma-separation applies to both the real and imaginary parts of the number, but a specified field width and alignment is applied to the whole of the resulting `1.5+3j` output. (Contributed by Eric Smith; bpo-1588 and bpo-7988.)

  The ‘F’ format code now always formats its output using
uppercase characters, so it will now produce ‘INF’ and ‘NAN’.
(Contributed by Eric Smith; bpo-3382.)

A low-level change: the `object.__format__() method now
triggers a `PendingDeprecationWarning` if it's passed a format
string, because the `__format__() method for `object` converts
the object to a string representation and formats that. Previously
the method silently applied the format string to the string
representation, but that could hide mistakes in Python code. If
you're supplying formatting information such as an alignment or
precision, presumably you're expecting the formatting to be
applied in some object-specific way. (Fixed by Eric Smith; bpo-
7994.)

- The `int()` and `long()` types gained a `bit_length` method
  that returns the number of bits necessary to represent its
  argument in binary:

  ```
  >>> n = 37
  >>> bin(n)
  '0b100101'
  >>> n.bit_length()
  6
  >>> n = 2**123-1
  >>> n.bit_length()
  123
  >>> (n+1).bit_length()
  124
  ```

  (Contributed by Fredrik Johansson and Victor Stinner; bpo-3439.)

- The `import` statement will no longer try an absolute import if a
  relative import (e.g. `from .os import sep`) fails. This fixes a
  bug, but could possibly break certain `import` statements that
were only working by accident. (Fixed by Meador Inge; bpo-7902.)

- It's now possible for a subclass of the built-in unicode type to override the `__unicode__()` method. (Implemented by Victor Stinner; bpo-1583863.)

- The `bytearray` type's `translate()` method now accepts `None` as its first argument. (Fixed by Georg Brandl; bpo-4759.)

- When using `@classmethod` and `@staticmethod` to wrap methods as class or static methods, the wrapper object now exposes the wrapped function as their `__func__` attribute. (Contributed by Amaury Forgeot d'Arc, after a suggestion by George Sakkis; bpo-5982.)

- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise `AttributeError` as you would expect. Fixed by Benjamin Peterson; bpo-7604.)

- Two new encodings are now supported: “cp720”, used primarily for Arabic text; and “cp858”, a variant of CP 850 that adds the euro symbol. (CP720 contributed by Alexander Belchenko and Amaury Forgeot d'Arc in bpo-1616979; CP858 contributed by Tim Hatch in bpo-8016.)

- The file object will now set the filename attribute on the `IOError` exception when trying to open a directory on POSIX platforms (noted by Jan Kaliszewski; bpo-4764), and now explicitly checks for and forbids writing to read-only file objects instead of trusting the C library to catch and report the error (fixed by Stefan Krah; bpo-5677).

- The Python tokenizer now translates line endings itself, so the
The `compile()` built-in function now accepts code using any line-ending convention. Additionally, it no longer requires that the code end in a newline.

- Extra parentheses in function definitions are illegal in Python 3.x, meaning that you get a syntax error from `def f((x)): pass`. In Python3-warning mode, Python 2.7 will now warn about this odd usage. (Noted by James Lingard; bpo-7362.)

- It's now possible to create weak references to old-style class objects. New-style classes were always weak-referenceable. (Fixed by Antoine Pitrou; bpo-8268.)

- When a module object is garbage-collected, the module's dictionary is now only cleared if no one else is holding a reference to the dictionary (bpo-7140).

### Interpreter Changes

A new environment variable, `PYTHONWARNINGS`, allows controlling warnings. It should be set to a string containing warning settings, equivalent to those used with the `-W` switch, separated by commas. (Contributed by Brian Curtin; bpo-7301.)

For example, the following setting will print warnings every time they occur, but turn warnings from the Cookie module into an error. (The exact syntax for setting an environment variable varies across operating systems and shells.)

```bash
export PYTHONWARNINGS=all,error::Cookie:0
```

### Optimizations
Several performance enhancements have been added:

- A new opcode was added to perform the initial setup for `with` statements, looking up the `__enter__()` and `__exit__()` methods. (Contributed by Benjamin Peterson.)

- The garbage collector now performs better for one common usage pattern: when many objects are being allocated without deallocating any of them. This would previously take quadratic time for garbage collection, but now the number of full garbage collections is reduced as the number of objects on the heap grows. The new logic only performs a full garbage collection pass when the middle generation has been collected 10 times and when the number of survivor objects from the middle generation exceeds 10% of the number of objects in the oldest generation. (Suggested by Martin von Löwis and implemented by Antoine Pitrou; bpo-4074.)

- The garbage collector tries to avoid tracking simple containers which can’t be part of a cycle. In Python 2.7, this is now true for tuples and dicts containing atomic types (such as ints, strings, etc.). Transitively, a dict containing tuples of atomic types won’t be tracked either. This helps reduce the cost of each garbage collection by decreasing the number of objects to be considered and traversed by the collector. (Contributed by Antoine Pitrou; bpo-4688.)

- Long integers are now stored internally either in base $2^{15}$ or in base $2^{30}$, the base being determined at build time. Previously, they were always stored in base $2^{15}$. Using base $2^{30}$ gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base $2^{30}$ on 64-bit machines.
and base $2^{15}$ on 32-bit machines; on Unix, there's a new configure option --enable-big-digits that can be used to override this default.

Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there's a new structseq sys.long_info that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

```python
>>> import sys
>>> sys.long_info
sys.long_info(bits_per_digit=30, sizeof_digit=4)
```

(Contributed by Mark Dickinson; bpo-4258.)

Another set of changes made long objects a few bytes smaller: 2 bytes smaller on 32-bit systems and 6 bytes on 64-bit. (Contributed by Mark Dickinson; bpo-5260.)

- The division algorithm for long integers has been made faster by tightening the inner loop, doing shifts instead of multiplications, and fixing an unnecessary extra iteration. Various benchmarks show speedups of between 50% and 150% for long integer divisions and modulo operations. (Contributed by Mark Dickinson; bpo-5512.) Bitwise operations are also significantly faster (initial patch by Gregory Smith; bpo-1087418).

- The implementation of % checks for the left-side operand being a Python string and special-cases it; this results in a 1–3% performance increase for applications that frequently use % with strings, such as templating libraries. (Implemented by Collin
List comprehensions with an if condition are compiled into faster bytecode. (Patch by Antoine Pitrou, back-ported to 2.7 by Jeffrey Yasskin; bpo-4715.)

Converting an integer or long integer to a decimal string was made faster by special-casing base 10 instead of using a generalized conversion function that supports arbitrary bases. (Patch by Gawain Bolton; bpo-6713.)

The split(), replace(), rindex(), rpartition(), and rsplit() methods of string-like types (strings, Unicode strings, and bytearray objects) now use a fast reverse-search algorithm instead of a character-by-character scan. This is sometimes faster by a factor of 10. (Added by Florent Xicluna; bpo-7462 and bpo-7622.)

The pickle and cPickle modules now automatically intern the strings used for attribute names, reducing memory usage of the objects resulting from unpickling. (Contributed by Jake McGuire; bpo-5084.)

The cPickle module now special-cases dictionaries, nearly halving the time required to pickle them. (Contributed by Collin Winter; bpo-5670.)
New and Improved Modules

As in every release, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The `bdb` module's base debugging class `Bdb` gained a feature for skipping modules. The constructor now takes an iterable containing glob-style patterns such as `django.*`; the debugger will not step into stack frames from a module that matches one of these patterns. (Contributed by Maru Newby after a suggestion by Senthil Kumaran; bpo-5142.)

- The `binascii` module now supports the buffer API, so it can be used with `memoryview` instances and other similar buffer objects. (Backported from 3.x by Florent Xicluna; bpo-7703.)

- Updated module: the `bsddb` module has been updated from 4.7.2devel9 to version 4.8.4 of the `pybsddb` package. The new version features better Python 3.x compatibility, various bug fixes, and adds several new BerkeleyDB flags and methods. (Updated by Jesús Cea Avión; bpo-8156. The pybsddb changelog can be read at http://hg.jcea.es/pybsddb/file/tip/ChangeLog.)

- The `bz2` module's `BZ2File` now supports the context management protocol, so you can write with `bz2.BZ2File(...) as f:`. (Contributed by Hagen Fürstenau; bpo-3860.)

- New class: the `Counter` class in the `collections` module is
useful for tallying data. Counter instances behave mostly like dictionaries but return zero for missing keys instead of raising a KeyError:

```python
>>> from collections import Counter
>>> c = Counter()
>>> for letter in 'here is a sample of english text'
...   c[letter] += 1
...
>>> c
Counter({'
': 6, 'e': 5, 's': 3, 'a': 2, 'i': 2, 'l': 2, 't': 2, 'g': 1, 'f': 1, 'm': 1, 'o': 1, 'n': 1, 'p': 1, 'r': 1, 'x': 1})
>>> c['e']
5
>>> c['z']
0
```

There are three additional Counter methods. most_common() returns the N most common elements and their counts. elements() returns an iterator over the contained elements, repeating each element as many times as its count. subtract() takes an iterable and subtracts one for each element instead of adding; if the argument is a dictionary or another Counter, the counts are subtracted.

```python
>>> c.most_common(5)
[(' ', 6), ('e', 5), ('s', 3), ('a', 2), ('i', 2)]
>>> c.elements() ->
'a', 'a', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'e', 'e', 'e', 'e', 'g', 'f', 'i', 'i', 'h', 'h', 'm', 'l', 'l', 'o', 'n', 'p', 's', 's', 's', 'r', 't', 't', 'x'
>>> c['e']
5
```
Contributed by Raymond Hettinger; bpo-1696199.

New class: `OrderedDict` is described in the earlier section PEP 372: Adding an Ordered Dictionary to collections.

New method: The `deque` data type now has a `count()` method that returns the number of contained elements equal to the supplied argument `x`, and a `reverse()` method that reverses the elements of the deque in-place. `deque` also exposes its maximum length as the read-only `maxlen` attribute. (Both features added by Raymond Hettinger.)

The `namedtuple` class now has an optional `rename` parameter. If `rename` is true, field names that are invalid because they've been repeated or aren't legal Python identifiers will be renamed to legal names that are derived from the field's position within the list of fields:

```python
>>> from collections import namedtuple
>>> T = namedtuple('T', ['field1', '$illegal', 'for'])
```

(Added by Raymond Hettinger; bpo-1818.)

Finally, the Mapping abstract base class now returns `NotImplemented` if a mapping is compared to another type that

```python
>>> c.subtract('very heavy on the letter e')
>>> c['e'] # Count is now lower
-1
```
isn't a Mapping. (Fixed by Daniel Stutzbach; bpo-8729.)

• Constructors for the parsing classes in the ConfigParser module now take an allow_no_value parameter, defaulting to false; if true, options without values will be allowed. For example:

```python
>>> import ConfigParser, StringIO
>>> sample_config = ""
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-bdb
... ""
>>> config = ConfigParser.RawConfigParser(allow_no_value)
>>> config.readfp(StringIO.StringIO(sample_config))
>>> config.get('mysqld', 'user')
'mysql'
>>> print config.get('mysqld', 'skip-bdb')
None
>>> print config.get('mysqld', 'unknown')
Traceback (most recent call last):
  ...
NoOptionError: No option 'unknown' in section: ''
```

(Contributed by Mats Kindahl; bpo-7005.)

• Deprecated function: `contextlib.nested()`, which allows handling more than one context manager with a single `with` statement, has been deprecated, because the `with` statement now supports multiple context managers.

• The `cookielib` module now ignores cookies that have an invalid version field, one that doesn't contain an integer value. (Fixed by John J. Lee; bpo-3924.)
The `copy` module's `deepcopy()` function will now correctly copy bound instance methods. (Implemented by Robert Collins; bpo-1515.)

The `ctypes` module now always converts `None` to a C NULL pointer for arguments declared as pointers. (Changed by Thomas Heller; bpo-4606.) The underlying `libffi` library has been updated to version 3.0.9, containing various fixes for different platforms. (Updated by Matthias Klose; bpo-8142.)

New method: the `datetime` module's `timedelta` class gained a `total_seconds()` method that returns the number of seconds in the duration. (Contributed by Brian Quinlan; bpo-5788.)

New method: the `Decimal` class gained a `from_float()` class method that performs an exact conversion of a floating-point number to a `Decimal`. This exact conversion strives for the closest decimal approximation to the floating-point representation's value; the resulting decimal value will therefore still include the inaccuracy, if any. For example, `Decimal.from_float(0.1)` returns `Decimal('0.10000000000000000555111512312578270211')` (Implemented by Raymond Hettinger; bpo-4796.)

Comparing instances of `Decimal` with floating-point numbers now produces sensible results based on the numeric values of the operands. Previously such comparisons would fall back to Python's default rules for comparing objects, which produced arbitrary results based on their type. Note that you still cannot combine `Decimal` and floating-point in other operations such as addition, since you should be explicitly choosing how to convert
between float and \texttt{Decimal}. (Fixed by Mark Dickinson; \texttt{bpo-2531}.)

The constructor for \texttt{Decimal} now accepts floating-point numbers (added by Raymond Hettinger; \texttt{bpo-8257}) and non-European Unicode characters such as Arabic-Indic digits (contributed by Mark Dickinson; \texttt{bpo-6595}).

Most of the methods of the \texttt{Context} class now accept integers as well as \texttt{Decimal} instances; the only exceptions are the \texttt{canonical()} and \texttt{is\_canonical()} methods. (Patch by Juan José Conti; \texttt{bpo-7633}.)

When using \texttt{Decimal} instances with a string's \texttt{format()} method, the default alignment was previously left-alignment. This has been changed to right-alignment, which is more sensible for numeric types. (Changed by Mark Dickinson; \texttt{bpo-6857}.)

Comparisons involving a signaling NaN value (or \texttt{sNAN}) now signal InvalidOperation instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or \texttt{NaN}) are now hashable. (Fixed by Mark Dickinson; \texttt{bpo-7279}.)

- The \texttt{difflib} module now produces output that is more compatible with modern \texttt{diff/patch} tools through one small change, using a tab character instead of spaces as a separator in the header giving the filename. (Fixed by Anatoly Techtonik; \texttt{bpo-7585}.)

- The Distutils \texttt{sdist} command now always regenerates the \texttt{MANIFEST} file, since even if the \texttt{MANIFEST.in} or \texttt{setup.py} files haven't been modified, the user might have created some
new files that should be included. (Fixed by Tarek Ziadé; bpo-8688.)

- The `doctest` module's `IGNORE_EXCEPTION_DETAIL` flag will now ignore the name of the module containing the exception being tested. (Patch by Lennart Regebro; bpo-7490.)

- The `email` module's `Message` class will now accept a Unicode-valued payload, automatically converting the payload to the encoding specified by `output_charset`. (Added by R. David Murray; bpo-1368247.)

- The `Fraction` class now accepts a single float or `Decimal` instance, or two rational numbers, as arguments to its constructor. (Implemented by Mark Dickinson; rationals added in bpo-5812, and float/decimal in bpo-8294.)

  Ordering comparisons (`<, <=, >, >=`) between fractions and complex numbers now raise a `TypeError`. This fixes an oversight, making the `Fraction` match the other numeric types.

- New class: `FTP_TLS` in the `ftplib` module provides secure FTP connections using TLS encapsulation of authentication as well as subsequent control and data transfers. (Contributed by Giampaolo Rodola; bpo-2054.)

  The `storbinary()` method for binary uploads can now restart uploads thanks to an added `rest` parameter (patch by Pablo Mouzo; bpo-6845.)

- New class decorator: `total_ordering()` in the `functools` module takes a class that defines an `__eq__()` method and one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, and
generates the missing comparison methods. Since the \_\_cmp\_\_() method is being deprecated in Python 3.x, this decorator makes it easier to define ordered classes. (Added by Raymond Hettinger; \textit{bpo-5479}.)

New function: \texttt{cmp\_to\_key()} will take an old-style comparison function that expects two arguments and return a new callable that can be used as the \textit{key} parameter to functions such as \texttt{sorted()}, \texttt{min()} and \texttt{max()}, etc. The primary intended use is to help with making code compatible with Python 3.x. (Added by Raymond Hettinger.)

- New function: the \texttt{gc} module's \texttt{is\_tracked()} returns true if a given instance is tracked by the garbage collector, false otherwise. (Contributed by Antoine Pitrou; \textit{bpo-4688}.)

- The \texttt{gzip} module's \texttt{GzipFile} now supports the context management protocol, so you can write with \texttt{gzip.GzipFile(...) as f:} (contributed by Hagen Fürstenau; \textit{bpo-3860}), and it now implements the \texttt{io.BufferedIOBase} ABC, so you can wrap it with \texttt{io.BufferedReader} for faster processing (contributed by Nir Aides; \textit{bpo-7471}). It's also now possible to override the modification time recorded in a gzipped file by providing an optional timestamp to the constructor. (Contributed by Jacques Frechet; \textit{bpo-4272}.)

Files in gzip format can be padded with trailing zero bytes; the \texttt{gzip} module will now consume these trailing bytes. (Fixed by Tadek Pietraszek and Brian Curtin; \textit{bpo-2846}.)

- New attribute: the \texttt{hashlib} module now has an algorithms
attribute containing a tuple naming the supported algorithms. In Python 2.7, `hashlib.algorithms` contains ('md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512'). (Contributed by Carl Chenet; bpo-7418.)

- The default HTTPResponse class used by the `httplib` module now supports buffering, resulting in much faster reading of HTTP responses. (Contributed by Kristján Valur Jónsson; bpo-4879.)

The `HTTPConnection` and `HTTPSConnection` classes now support a `source_address` parameter, a `(host, port)` 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; bpo-3972.)

- The `ihooks` module now supports relative imports. Note that `ihooks` is an older module for customizing imports, superseded by the `imputil` module added in Python 2.0. (Relative import support added by Neil Schemenauer.)

- The `imaplib` module now supports IPv6 addresses. (Contributed by Derek Morr; bpo-1655.)

- New function: the `inspect` module's `getcallargs()` takes a callable and its positional and keyword arguments, and figures out which of the callable's parameters will receive each argument, returning a dictionary mapping argument names to their values. For example:

```python
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'b': 2, 'pos': (3,), 'named': {}}
```
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'b': 1, 'pos': (), 'named': {'x': 4}}
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() takes at least 1 argument (0 given)

Contributed by George Sakkis; bpo-3135.

- Updated module: The io library has been upgraded to the version shipped with Python 3.1. For 3.1, the I/O library was entirely rewritten in C and is 2 to 20 times faster depending on the task being performed. The original Python version was renamed to the _pyio module.

One minor resulting change: the io.TextIOBase class now has an errors attribute giving the error setting used for encoding and decoding errors (one of 'strict', 'replace', 'ignore').

The io.FileIO class now raises an OSError when passed an invalid file descriptor. (Implemented by Benjamin Peterson; bpo-4991.) The truncate() method now preserves the file position; previously it would change the file position to the end of the new file. (Fixed by Pascal Chambon; bpo-6939.)

- New function: itertools.compress(data, selectors) takes two iterators. Elements of data are returned if the corresponding value in selectors is true:

```
itertools.compress('ABCDEF', [1,0,1,0,1,1]) => A, C, E, F
```

New function:
**itertools.combinations_with_replacement(iter, r)** returns all the possible \( r \)-length combinations of elements from the iterable \( \text{iter} \). Unlike **combinations()**, individual elements can be repeated in the generated combinations:

```python
itertools.combinations_with_replacement('abc', 2)
('a', 'a'), ('a', 'b'), ('a', 'c'),
('b', 'b'), ('b', 'c'), ('c', 'c')
```

Note that elements are treated as unique depending on their position in the input, not their actual values.

The **itertools.count()** function now has a **step argument** that allows incrementing by values other than 1. **count()** also now allows keyword arguments, and using non-integer values such as floats or **Decimal** instances. (Implemented by Raymond Hettinger; **bpo-5032**.)

**itertools.combinations()** and **itertools.product()** previously raised **ValueError** for values of \( r \) larger than the input iterable. This was deemed a specification error, so they now return an empty iterator. (Fixed by Raymond Hettinger; **bpo-4816**.)

- Updated module: The **json** module was upgraded to version 2.0.9 of the simplejson package, which includes a C extension that makes encoding and decoding faster. (Contributed by Bob Ippolito; **bpo-4136**.)

To support the new **collections.OrderedDict** type, **json.load()** now has an optional **object_pairs_hook** parameter that will be called with any object literal that decodes to a list of pairs. (Contributed by Raymond Hettinger; **bpo-5381**.)
• The `mailbox` module's `Maildir` class now records the timestamp on the directories it reads, and only re-reads them if the modification time has subsequently changed. This improves performance by avoiding unneeded directory scans. (Fixed by A.M. Kuchling and Antoine Pitrou; bpo-1607951, bpo-6896.)

• New functions: the `math` module gained `erf()` and `erfc()` for the error function and the complementary error function, `expm1()` which computes $e^{x} - 1$ with more precision than using `exp()` and subtracting 1, `gamma()` for the Gamma function, and `lgamma()` for the natural log of the Gamma function. (Contributed by Mark Dickinson and nirinA raseliarison; bpo-3366.)

• The `multiprocessing` module's `Manager*` classes can now be passed a callable that will be called whenever a subprocess is started, along with a set of arguments that will be passed to the callable. (Contributed by lekma; bpo-5585.)

  The Pool class, which controls a pool of worker processes, now has an optional `maxtasksperchild` parameter. Worker processes will perform the specified number of tasks and then exit, causing the Pool to start a new worker. This is useful if tasks may leak memory or other resources, or if some tasks will cause the worker to become very large. (Contributed by Charles Cazabon; bpo-6963.)

• The `nntplib` module now supports IPv6 addresses. (Contributed by Derek Morr; bpo-1664.)

• New functions: the `os` module wraps the following POSIX system calls: `getresgid()` and `getresuid()`, which return the real,
effective, and saved GIDs and UIDs; \texttt{setresgid()} and \texttt{setresuid()}, which set real, effective, and saved GIDs and UIDs to new values; \texttt{initgroups()}, which initialize the group access list for the current process. (GID/UID functions contributed by Travis H.; \texttt{bpo-6508}. Support for initgroups added by Jean-Paul Calderone; \texttt{bpo-7333}.)

The \texttt{os.fork()} function now re-initializes the import lock in the child process; this fixes problems on Solaris when \texttt{fork()} is called from a thread. (Fixed by Zsolt Cserna; \texttt{bpo-7242}.)

- In the \texttt{os.path} module, the \texttt{normpath()} and \texttt{abspath()} functions now preserve Unicode; if their input path is a Unicode string, the return value is also a Unicode string. (\texttt{normpath()} fixed by Matt Giuca in \texttt{bpo-5827}; \texttt{abspath()} fixed by Ezio Melotti in \texttt{bpo-3426}.)

- The \texttt{pydoc} module now has help for the various symbols that Python uses. You can now do \texttt{help('<<')} or \texttt{help('@')}, for example. (Contributed by David Laban; \texttt{bpo-4739}.)

- The \texttt{re} module’s \texttt{split()}, \texttt{sub()}, and \texttt{subn()} now accept an optional \texttt{flags} argument, for consistency with the other functions in the module. (Added by Gregory P. Smith.)

- New function: \texttt{run_path()} in the \texttt{runpy} module will execute the code at a provided \texttt{path} argument. \texttt{path} can be the path of a Python source file (\texttt{example.py}), a compiled bytecode file (\texttt{example.pyc}), a directory (\texttt{./package/}), or a zip archive (\texttt{example.zip}). If a directory or zip path is provided, it will be added to the front of \texttt{sys.path} and the module \texttt{__main__} will be imported. It’s expected that the directory or zip contains a
__main__.py; if it doesn't, some other __main__.py might be imported from a location later in sys.path. This makes more of the machinery of runpy available to scripts that want to mimic the way Python's command line processes an explicit path name. (Added by Nick Coghlan; bpo-6816.)

- New function: in the shutil module, make_archive() takes a filename, archive type (zip or tar-format), and a directory path, and creates an archive containing the directory's contents. (Added by Tarek Ziadé.)

shutil's copyfile() and copytree() functions now raise a SpecialFileError exception when asked to copy a named pipe. Previously the code would treat named pipes like a regular file by opening them for reading, and this would block indefinitely. (Fixed by Antoine Pitrou; bpo-3002.)

- The signal module no longer re-installs the signal handler unless this is truly necessary, which fixes a bug that could make it impossible to catch the EINTR signal robustly. (Fixed by Charles-Francois Natali; bpo-8354.)

- New functions: in the site module, three new functions return various site- and user-specific paths. getsitepackages() returns a list containing all global site-packages directories, getusersitepackages() returns the path of the user's site-packages directory, and getuserbase() returns the value of the USER_BASE environment variable, giving the path to a directory that can be used to store data. (Contributed by Tarek Ziadé; bpo-6693.)

The site module now reports exceptions occurring when the
sitecustomize module is imported, and will no longer catch and swallow the `KeyboardInterrupt` exception. (Fixed by Victor Stinner; bpo-3137.)

- The `create_connection()` function gained a `source_address` parameter, a `(host, port)` 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; bpo-3972.)

The `recv_into()` and `recvfrom_into()` methods will now write into objects that support the buffer API, most usefully the `bytearray` and `memoryview` objects. (Implemented by Antoine Pitrou; bpo-8104.)

- The `SocketServer` module's `TCPServer` class now supports socket timeouts and disabling the Nagle algorithm. The `disable_nagle_algorithm` class attribute defaults to False; if overridden to be true, new request connections will have the TCP_NODELAY option set to prevent buffering many small sends into a single TCP packet. The `timeout` class attribute can hold a timeout in seconds that will be applied to the request socket; if no request is received within that time, `handle_timeout()` will be called and `handle_request()` will return. (Contributed by Kristján Valur Jónsson; bpo-6192 and bpo-6267.)

- Updated module: the `sqlite3` module has been updated to version 2.6.0 of the `pysqlite` package. Version 2.6.0 includes a number of bugfixes, and adds the ability to load SQLite extensions from shared libraries. Call the `enable_load_extension(True)` method to enable extensions, and then call `load_extension()` to load a particular shared library. (Updated by Gerhard Häring.)
The **ssl** module’s **SSLSocket** objects now support the buffer API, which fixed a test suite failure (fix by Antoine Pitrou; bpo-7133) and automatically set OpenSSL’s **SSL_MODE_AUTO_RETRY**, which will prevent an error code being returned from `recv()` operations that trigger an SSL renegotiation (fix by Antoine Pitrou; bpo-8222).

The **ssl.wrap_socket()** constructor function now takes a *ciphers* argument that’s a string listing the encryption algorithms to be allowed; the format of the string is described in the OpenSSL documentation. (Added by Antoine Pitrou; bpo-8322.)

Another change makes the extension load all of OpenSSL’s ciphers and digest algorithms so that they’re all available. Some SSL certificates couldn’t be verified, reporting an “unknown algorithm” error. (Reported by Beda Kosata, and fixed by Antoine Pitrou; bpo-8484.)

The version of OpenSSL being used is now available as the module attributes **ssl.OPENSSL_VERSION** (a string), **ssl.OPENSSL_VERSION_INFO** (a 5-tuple), and **ssl.OPENSSL_VERSION_NUMBER** (an integer). (Added by Antoine Pitrou; bpo-8321.)

The **struct** module will no longer silently ignore overflow errors when a value is too large for a particular integer format code (one of `bBhIILlQQ`); it now always raises a **struct.error** exception. (Changed by Mark Dickinson; bpo-1523.) The **pack()** function will also attempt to use **__index__()** to convert and pack non-integers before trying the **__int__()** method or reporting an error. (Changed by Mark Dickinson; bpo-8300.)
• New function: the subprocess module's check_output() runs a command with a specified set of arguments and returns the command's output as a string when the command runs without error, or raises a CalledProcessError exception otherwise.

```python
>>> subprocess.check_output(['df', '-h', '.'])
'Filesystem  Size  Used  Avail  Capacity  Mounted
/dev/disk0s2  52G   49G   3.0G    94%   /
'
```

(Contributed by Gregory P. Smith.)

The subprocess module will now retry its internal system calls on receiving an EINTR signal. (Reported by several people; final patch by Gregory P. Smith in bpo-1068268.)

• New function: is_declared_global() in the symtable module returns true for variables that are explicitly declared to be global, false for ones that are implicitly global. (Contributed by Jeremy Hylton.)

• The syslog module will now use the value of sys.argv[0] as the identifier instead of the previous default value of 'python'. (Changed by Sean Reifschneider; bpo-8451.)

• The sys.version_info value is now a named tuple, with attributes named major, minor, micro, releaselevel, and serial. (Contributed by Ross Light; bpo-4285.)

sys.getwindowsversion() also returns a named tuple, with
attributes named major, minor, build, platform, service_pack, service_pack_major, service_pack_minor, suite_mask, and product_type. (Contributed by Brian Curtin; bpo-7766.)

- The `tarfile` module’s default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; bpo-7357.)

  `tarfile` now supports filtering the `TarInfo` objects being added to a tar file. When you call `add()`, you may supply an optional `filter` argument that's a callable. The `filter` callable will be passed the `TarInfo` for every file being added, and can modify and return it. If the callable returns `None`, the file will be excluded from the resulting archive. This is more powerful than the existing `exclude` argument, which has therefore been deprecated. (Added by Lars Gustäbel; bpo-6856.) The `TarFile` class also now supports the context management protocol. (Added by Lars Gustäbel; bpo-7232.)

- The `wait()` method of the `threading.Event` class now returns the internal flag on exit. This means the method will usually return true because `wait()` is supposed to block until the internal flag becomes true. The return value will only be false if a timeout was provided and the operation timed out. (Contributed by Tim Lesher; bpo-1674032.)

- The Unicode database provided by the `unicodedata` module is
now used internally to determine which characters are numeric, whitespace, or represent line breaks. The database also includes information from the Unihan.txt data file (patch by Anders Chrigström and Amaury Forgeot d'Arc; bpo-1571184) and has been updated to version 5.2.0 (updated by Florent Xicluna; bpo-8024).

- The urlparse module’s urlsplit() now handles unknown URL schemes in a fashion compliant with RFC 3986: if the URL is of the form "<something>://...", the text before the :// is treated as the scheme, even if it’s a made-up scheme that the module doesn’t know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', '', '//host/filename?query', '', '')
```

Python 2.7 (and Python 2.6.5) will return:

```python
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', 'host', '/filename?query', '', '')
```

(Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

The urlparse module also supports IPv6 literal addresses as defined by RFC 2732 (contributed by Senthil Kumaran; bpo-2987).
New class: the `WeakSet` class in the `weakref` module is a set that only holds weak references to its elements; elements will be removed once there are no references pointing to them. (Originally implemented in Python 3.x by Raymond Hettinger, and backported to 2.7 by Michael Foord.)

The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!--comment -->`). (Patch by Neil Muller; bpo-2746.)

The XML-RPC client and server, provided by the `xmlrpclib` and `SimpleXMLRPCServer` modules, have improved performance by supporting HTTP/1.1 keep-alive and by optionally using gzip encoding to compress the XML being exchanged. The gzip compression is controlled by the `encode_threshold` attribute of `SimpleXMLRPCRequestHandler`, which contains a size in bytes; responses larger than this will be compressed. (Contributed by Kristján Valur Jónsson; bpo-6267.)

The `zipfile` module's `ZipFile` now supports the context management protocol, so you can write with `zipfile.ZipFile(...) as f:`. (Contributed by Brian Curtin; bpo-5511.)

`zipfile` now also supports archiving empty directories and extracts them correctly. (Fixed by Kuba Wieczorek; bpo-4710.)
Reading files out of an archive is faster, and interleaving `read()` and `readline()` now works correctly. (Contributed by Nir Aides; bpo-7610.)

The `is_zipfile()` function now accepts a file object, in addition to the path names accepted in earlier versions. (Contributed by Gabriel Genellina; bpo-4756.)

The `writestr()` method now has an optional `compress_type` parameter that lets you override the default compression method specified in the `ZipFile` constructor. (Contributed by Ronald Oussoren; bpo-6003.)

New module: importlib

Python 3.1 includes the `importlib` package, a re-implementation of the logic underlying Python’s `import` statement. `importlib` is useful for implementors of Python interpreters and to users who wish to write new importers that can participate in the import process. Python 2.7 doesn’t contain the complete `importlib` package, but instead has a tiny subset that contains a single function, `import_module()`.

`import_module(name, package=None)` imports a module. `name` is a string containing the module or package’s name. It’s possible to do relative imports by providing a string that begins with a `.` character, such as `..utils.errors`. For relative imports, the `package` argument must be provided and is the name of the package that will be used as the anchor for the relative import. `import_module()` both inserts the imported module into `sys.modules` and returns the module object.
Here are some examples:

```python
>>> from importlib import import_module
>>> anydbm = import_module('anydbm')  # Standard absolute import
>>> anydbm
<module 'anydbm' from '/p/python/Lib/anydbm.py'>
>>> # Relative import
>>> file_util = import_module('..file_util', 'distutils.command')
>>> file_util
<module 'distutils.file_util' from '/python/Lib/distutils/file_util.pyc'>
```

`importlib` was implemented by Brett Cannon and introduced in Python 3.1.

**New module: sysconfig**

The `sysconfig` module has been pulled out of the Distutils package, becoming a new top-level module in its own right. `sysconfig` provides functions for getting information about Python’s build process: compiler switches, installation paths, the platform name, and whether Python is running from its source directory.

Some of the functions in the module are:

- `get_config_var()` returns variables from Python’s Makefile and the `pyconfig.h` file.
- `get_config_vars()` returns a dictionary containing all of the configuration variables.
- `get_path()` returns the configured path for a particular type of module: the standard library, site-specific modules, platform-specific modules, etc.
- `is_python_build()` returns true if you’re running a binary from a Python source tree, and false otherwise.
Consult the sysconfig documentation for more details and for a complete list of functions.

The Distutils package and sysconfig are now maintained by Tarek Ziadé, who has also started a Distutils2 package (source repository at https://hg.python.org/distutils2/) for developing a next-generation version of Distutils.

**ttk: Themed Widgets for Tk**

Tcl/Tk 8.5 includes a set of themed widgets that re-implement basic Tk widgets but have a more customizable appearance and can therefore more closely resemble the native platform's widgets. This widget set was originally called Tile, but was renamed to Ttk (for “themed Tk”) on being added to Tcl/Tk release 8.5.

To learn more, read the ttk module documentation. You may also wish to read the Tcl/Tk manual page describing the Ttk theme engine, available at https://www.tcl.tk/man/tcl8.5/TkCmd/ttk_intro.htm. Some screenshots of the Python/Ttk code in use are at https://code.google.com/archive/p/python-ttk/wikis/Screenshots.wiki.

The ttk module was written by Guilherme Polo and added in bpo-2983. An alternate version called Tile.py, written by Martin Franklin and maintained by Kevin Walzer, was proposed for inclusion in bpo-2618, but the authors argued that Guilherme Polo’s work was more comprehensive.

**Updated module: unittest**

The unittest module was greatly enhanced; many new features were added. Most of these features were implemented by Michael
The enhanced version of the module is downloadable separately for use with Python versions 2.4 to 2.6, packaged as the unittest2 package, from https://pypi.org/project/unittest2.

When used from the command line, the module can automatically discover tests. It’s not as fancy as py.test or nose, but provides a simple way to run tests kept within a set of package directories. For example, the following command will search the test/ subdirectory for any importable test files named test*.py:

```
python -m unittest discover -s test
```

Consult the unittest module documentation for more details.
(Developed in bpo-6001.)

The main() function supports some other new options:

- `-b` or `--buffer` will buffer the standard output and standard error streams during each test. If the test passes, any resulting output will be discarded; on failure, the buffered output will be displayed.

- `-c` or `--catch` will cause the control-C interrupt to be handled more gracefully. Instead of interrupting the test process immediately, the currently running test will be completed and then the partial results up to the interruption will be reported. If you’re impatient, a second press of control-C will cause an immediate interruption.

This control-C handler tries to avoid causing problems when the code being tested or the tests being run have defined a signal handler of their own, by noticing that a signal handler was already
set and calling it. If this doesn't work for you, there's a
removeHandler() decorator that can be used to mark tests that
should have the control-C handling disabled.

- -f or --failfast makes test execution stop immediately when
a test fails instead of continuing to execute further tests.
(Suggested by Cliff Dyer and implemented by Michael Foord; bpo-
8074.)

The progress messages now show ‘x’ for expected failures and ‘u’ for
unexpected successes when run in verbose mode. (Contributed by
Benjamin Peterson.)

Test cases can raise the SkipTest exception to skip a test (bpo-
1034053).

The error messages for assertEqual(), assertTrue(), and
assertFalse() failures now provide more information. If you set the
longMessage attribute of your TestCase classes to true, both the
standard error message and any additional message you provide will
be printed for failures. (Added by Michael Foord; bpo-5663.)

The assertRaises() method now returns a context handler when
called without providing a callable object to run. For example, you can
call this:

```python
with self.assertRaises(KeyError):
    {}['foo']
```

(Implemented by Antoine Pitrou; bpo-4444.)

Module- and class-level setup and teardown fixtures are now
supported. Modules can contain setUpModule() and
tearDownModule() functions. Classes can have setUpClass() and tearDownClass() methods that must be defined as class methods (using @classmethod or equivalent). These functions and methods are invoked when the test runner switches to a test case in a different module or class.

The methods addCleanup() and doCleanups() were added. addCleanup() lets you add cleanup functions that will be called unconditionally (after setUp() if setUp() fails, otherwise after tearDown()). This allows for much simpler resource allocation and deallocation during tests (bpo-5679).

A number of new methods were added that provide more specialized tests. Many of these methods were written by Google engineers for use in their test suites; Gregory P. Smith, Michael Foord, and GvR worked on merging them into Python's version of unittest.

- **assertIsNone()** and **assertIsNotNone()** take one expression and verify that the result is or is not None.
- **assertIs()** and **assertIsNot()** take two values and check whether the two values evaluate to the same object or not. (Added by Michael Foord; bpo-2578.)
- **assertIsInstance()** and **assertNotIsInstance()** check whether the resulting object is an instance of a particular class, or of one of a tuple of classes. (Added by Georg Brandl; bpo-7031.)
- **assertGreater(), assertGreaterEqual(), assertLess(), and assertLessEqual()** compare two quantities.
- **assertMultiLineEqual()** compares two strings, and if they're not equal, displays a helpful comparison that highlights the differences in the two strings. This comparison is now used by default when Unicode strings are compared with assertEqual().
assertRegexpMatches() and assertNotRegexpMatches() checks whether the first argument is a string matching or not matching the regular expression provided as the second argument (bpo-8038).

assertRaisesRegexp() checks whether a particular exception is raised, and then also checks that the string representation of the exception matches the provided regular expression.

assertIn() and assertNotIn() tests whether first is or is not in second.

assertItemsEqual() tests whether two provided sequences contain the same elements.

assertSetEqual() compares whether two sets are equal, and only reports the differences between the sets in case of error.

Similarly, assertListEqual() and assertTupleEqual() compare the specified types and explain any differences without necessarily printing their full values; these methods are now used by default when comparing lists and tuples using assertEqual(). More generally, assertSequenceEqual() compares two sequences and can optionally check whether both sequences are of a particular type.

assertDictEqual() compares two dictionaries and reports the differences; it's now used by default when you compare two dictionaries using assertEqual().

assertDictContainsSubset() checks whether all of the key/value pairs in first are found in second.

assertAlmostEqual() and assertNotAlmostEqual() test whether first and second are approximately equal. This method can either round their difference to an optionally-specified number of places (the default is 7) and compare it to zero, or require the difference to be smaller than a supplied delta value.

loadTestsFromName() properly honors the suiteClass attribute of the TestLoader. (Fixed by Mark Roddy; bpo-6866.)

A new hook lets you extend the assertEqual() method to
handle new data types. The `addTypeEqualityFunc()` method takes a type object and a function. The function will be used when both of the objects being compared are of the specified type. This function should compare the two objects and raise an exception if they don't match; it's a good idea for the function to provide additional information about why the two objects aren't matching, much as the new sequence comparison methods do.

`unittest.main()` now takes an optional `exit` argument. If false, `main()` doesn't call `sys.exit()`, allowing `main()` to be used from the interactive interpreter. (Contributed by J. Pablo Fernández; bpo-3379.)

`TestResult` has new `startTestRun()` and `stopTestRun()` methods that are called immediately before and after a test run. (Contributed by Robert Collins; bpo-5728.)

With all these changes, the `unittest.py` was becoming awkwardly large, so the module was turned into a package and the code split into several files (by Benjamin Peterson). This doesn't affect how the module is imported or used.

See also:

http://www.voidspace.org.uk/python/articles/unittest2.shtml

Describes the new features, how to use them, and the rationale for various design decisions. (By Michael Foord.)

Updated module: `ElementTree 1.3`

The version of the `ElementTree` library included with Python was updated to version 1.3. Some of the new features are:
- The various parsing functions now take a `parser` keyword argument giving an `XMLParser` instance that will be used. This makes it possible to override the file's internal encoding:

```python
p = ET.XMLParser(encoding='utf-8')
t = ET.XML("""<root/>"""", parser=p)
```

Errors in parsing XML now raise a `ParseError` exception, whose instances have a position attribute containing a `(line, column)` tuple giving the location of the problem.

- ElementTree's code for converting trees to a string has been significantly reworked, making it roughly twice as fast in many cases. The `ElementTree.write()` and `Element.write()` methods now have a `method` parameter that can be “xml” (the default), “html”, or “text”. HTML mode will output empty elements as `<empty></empty>` instead of `<empty/>`, and text mode will skip over elements and only output the text chunks. If you set the tag attribute of an element to `None` but leave its children in place, the element will be omitted when the tree is written out, so you don’t need to do more extensive rearrangement to remove a single element.

Namespace handling has also been improved. All `xmlns:<whatever>` declarations are now output on the root element, not scattered throughout the resulting XML. You can set the default namespace for a tree by setting the `default_namespace` attribute and can register new prefixes with `register_namespace()`. In XML mode, you can use the `true/false xml_declaration` parameter to suppress the XML declaration.
• New Element method: `extend()` appends the items from a sequence to the element's children. Elements themselves behave like sequences, so it's easy to move children from one element to another:

```python
from xml.etree import ElementTree as ET

t = ET.XML('"""<list>
   <item>1</item> <item>2</item> <item>3</item>
</list>"""
new = ET.XML('<root/>
new.extend(t)

# Outputs <root><item>1</item>...</root>
print ET.tostring(new)
```

• New Element method: `iter()` yields the children of the element as a generator. It's also possible to write `for child in elem:` to loop over an element's children. The existing method `getiterator()` is now deprecated, as is `getchildren()` which constructs and returns a list of children.

• New Element method: `itertext()` yields all chunks of text that are descendants of the element. For example:

```python
t = ET.XML('"""<list>
   <item>1</item> <item>2</item> <item>3</item>
</list>"""

# Outputs ['
    ', '1', ' ', '2', ' ', '3', ' 
print list(t.itertext())
```

• Deprecated: using an element as a Boolean (i.e., `if elem:`)
would return true if the element had any children, or false if there were no children. This behaviour is confusing – None is false, but so is a childless element? – so it will now trigger a FutureWarning. In your code, you should be explicit: write `len(elem) != 0` if you're interested in the number of children, or `elem is not None`.

Fredrik Lundh develops ElementTree and produced the 1.3 version; you can read his article describing 1.3 at [http://effbot.org/zone/elementtree-13-intro.htm](http://effbot.org/zone/elementtree-13-intro.htm). Florent Xicluna updated the version included with Python, after discussions on python-dev and in [bpo-6472](http://bugs.python.org/issue6472).
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The latest release of the GNU Debugger, GDB 7, can be **scripted using Python**. When you begin debugging an executable program P, GDB will look for a file named `P-gdb.py` and automatically read it. Dave Malcolm contributed a `python-gdb.py` that adds a number of commands useful when debugging Python itself. For example, `py-up` and `py-down` go up or down one Python stack frame, which usually corresponds to several C stack frames. `py-print` prints the value of a Python variable, and `py-bt` prints the Python stack trace. (Added as a result of bpo-8032.)

- If you use the `.gdbinit` file provided with Python, the “pyo” macro in the 2.7 version now works correctly when the thread being debugged doesn't hold the GIL; the macro now acquires it before printing. (Contributed by Victor Stinner; bpo-3632.)

- `Py_AddPendingCall()` is now thread-safe, letting any worker thread submit notifications to the main Python thread. This is particularly useful for asynchronous IO operations. (Contributed by Kristján Valur Jónsson; bpo-4293.)

- New function: `PyCode_NewEmpty()` creates an empty code object; only the filename, function name, and first line number are required. This is useful for extension modules that are attempting to construct a more useful traceback stack. Previously such extensions needed to call `PyCode_New()`, which had many more arguments. (Added by Jeffrey Yasskin.)

- New function: `PyErr_NewExceptionWithDoc()` creates a new
exception class, just as the existing `PyErr_NewException()`
does, but takes an extra `char*` argument containing the
docstring for the new exception class. (Added by ‘lekma’ on the
Python bug tracker; [bpo-7033](https://bugs.python.org/issue7033).)

- New function: `PyFrame_GetLineNumber()` takes a frame
  object and returns the line number that the frame is currently
  executing. Previously code would need to get the index of the
  bytecode instruction currently executing, and then look up the line
  number corresponding to that address. (Added by Jeffrey
  Yasskin.)

- New functions: `PyLong_AsLongAndOverflow()` and
  `PyLong_AsLongLongAndOverflow()` approximates a Python
  long integer as a C `long` or `long long`. If the number is too
  large to fit into the output type, an `overflow` flag is set and returned
to the caller. (Contributed by Case Van Horsen; [bpo-7528](https://bugs.python.org/issue7528) and
[bpo-7767](https://bugs.python.org/issue7767).)

- New function: stemming from the rewrite of string-to-float
  conversion, a new `PyOS_string_to_double()` function was
  added. The old `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` functions are now deprecated.

- New function: `PySys_SetArgvEx()` sets the value of
  `sys.argv` and can optionally update `sys.path` to include the
directory containing the script named by `sys.argv[0]`
depending on the value of an `updatepath` parameter.

This function was added to close a security hole for applications
that embed Python. The old function, `PySys_SetArgv()`, would
always update `sys.path`, and sometimes it would add the
current directory. This meant that, if you ran an application embedding Python in a directory controlled by someone else, attackers could put a Trojan-horse module in the directory (say, a file named `os.py`) that your application would then import and run.

If you maintain a C/C++ application that embeds Python, check whether you’re calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with `updatepath` set to false.

Security issue reported as CVE-2008-5983; discussed in bpo-5753, and fixed by Antoine Pitrou.

- New macros: the Python header files now define the following macros: `Py_ISALNUM`, `Py_ISALPHA`, `Py_ISDIGIT`, `Py_ISLOWER`, `Py_ISSPACE`, `Py_ISUPPER`, `Py_ISXDIGIT`, `Py_TOLower`, and `Py_TOUPPER`. All of these functions are analogous to the C standard macros for classifying characters, but ignore the current locale setting, because in several places Python needs to analyze characters in a locale-independent way. (Added by Eric Smith; bpo-5793.)

- Removed function: `PyEval_CallObject` is now only available as a macro. A function version was being kept around to preserve ABI linking compatibility, but that was in 1997; it can certainly be deleted by now. (Removed by Antoine Pitrou; bpo-8276.)

- New format codes: the `PyFormat_FromString()`, `PyFormat_FromStringV()`, and `PyErr_Format()` functions now accept `%lld` and `%llu` format codes for displaying C’s `long long` types. (Contributed by Mark Dickinson; bpo-7228.)
The complicated interaction between threads and process forking has been changed. Previously, the child process created by `os.fork()` might fail because the child is created with only a single thread running, the thread performing the `os.fork()`. If other threads were holding a lock, such as Python’s import lock, when the fork was performed, the lock would still be marked as “held” in the new process. But in the child process nothing would ever release the lock, since the other threads weren’t replicated, and the child process would no longer be able to perform imports.

Python 2.7 acquires the import lock before performing an `os.fork()`, and will also clean up any locks created using the `threading` module. C extension modules that have internal locks, or that call `fork()` themselves, will not benefit from this clean-up.

(Fixed by Thomas Wouters; bpo-1590864.)

The `Py_Finalize()` function now calls the internal `threading._shutdown()` function; this prevents some exceptions from being raised when an interpreter shuts down.

(Patch by Adam Olsen; bpo-1722344.)

When using the `PyMemberDef` structure to define attributes of a type, Python will no longer let you try to delete or set a `T_STRING_INPLACE` attribute.

Global symbols defined by the `ctypes` module are now prefixed with `Py`, or with `_ctypes`. (Implemented by Thomas Heller; bpo-3102.)

New configure option: the `--with-system-expat` switch allows
building the pyexpat module to use the system Expat library. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-7609.)

- New configure option: the --with-valgrind option will now disable the pymalloc allocator, which is difficult for the Valgrind memory-error detector to analyze correctly. Valgrind will therefore be better at detecting memory leaks and overruns. (Contributed by James Henstridge; bpo-2422.)

- New configure option: you can now supply an empty string to --with-dbmliborder= in order to disable all of the various DBM modules. (Added by Arfrever Frehtes Taifersar Arahesis; bpo-6491.)

- The configure script now checks for floating-point rounding bugs on certain 32-bit Intel chips and defines a X87_DOUBLE_ROUNDING preprocessor definition. No code currently uses this definition, but it’s available if anyone wishes to use it. (Added by Mark Dickinson; bpo-2937.)

  configure also now sets a LDCXXSHARED Makefile variable for supporting C++ linking. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-1222585.)

- The build process now creates the necessary files for pkg-config support. (Contributed by Clinton Roy; bpo-3585.)

- The build process now supports Subversion 1.7. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-6094.)

**Capsules**

Python 3.1 adds a new C datatype, PyCapsule, for providing a C API
to an extension module. A capsule is essentially the holder of a C
\texttt{void *} pointer, and is made available as a module attribute; for
example, the \texttt{socket} module's API is exposed as \texttt{socket.CAPI},
and \texttt{unicodedata} exposes \texttt{ucnhash\_CAPI}. Other extensions can
import the module, access its dictionary to get the capsule object, and
then get the \texttt{void *} pointer, which will usually point to an array of
pointers to the module's various API functions.

There is an existing data type already used for this, \texttt{PyCObject}, but it
doesn't provide type safety. Evil code written in pure Python could
cause a segmentation fault by taking a \texttt{PyCObject} from module A
and somehow substituting it for the \texttt{PyCObject} in module B. Capsules know their own name, and getting the pointer requires
providing the name:

\begin{verbatim}
void *vtable;

if (!PyCapsule_IsValid(capsule, "mymodule.CAPI") {
    PyErr_SetString(PyExc_ValueError, "argument
type invalid");
    return NULL;
}

vtable = PyCapsule_GetPointer(capsule, "mymodule.CAPI");
\end{verbatim}

You are assured that \texttt{vtable} points to whatever you're expecting. If a
different capsule was passed in, \texttt{PyCapsule_IsValid()} would
detect the mismatched name and return false. Refer to \texttt{Providing a C
API for an Extension Module} for more information on using these
objects.

Python 2.7 now uses capsules internally to provide various extension-
module APIs, but the \texttt{PyCObject\_AsVoidPtr()} was modified to
handle capsules, preserving compile-time compatibility with the CObject interface. Use of PyCObject_AsVoidPtr() will signal a PendingDeprecationWarning, which is silent by default.

Implemented in Python 3.1 and backported to 2.7 by Larry Hastings; discussed in bpo-5630.

Port-Specific Changes: Windows

- The msvcrt module now contains some constants from the crtassem.h header file: CRT_ASSEMBLY_VERSION, VC_ASSEMBLY_PUBLICKEYTOKEN, and LIBRARIES_ASSEMBLY_NAME_PREFIX. (Contributed by David Cournapeau; bpo-4365.)
- The _winreg module for accessing the registry now implements the CreateKeyEx() and DeleteKeyEx() functions, extended versions of previously-supported functions that take several extra arguments. The DisableReflectionKey(), EnableReflectionKey(), and QueryReflectionKey() were also tested and documented. (Implemented by Brian Curtin: bpo-7347.)
- The new _beginthreadex() API is used to start threads, and the native thread-local storage functions are now used. (Contributed by Kristján Valur Jónsson; bpo-3582.)
- The os.kill() function now works on Windows. The signal value can be the constants CTRL_C_EVENT, CTRL_BREAK_EVENT, or any integer. The first two constants will send Control-C and Control-Break keystroke events to subprocesses; any other value will use the TerminateProcess() API. (Contributed by Miki Tebeka; bpo-1220212.)
- The os.listdir() function now correctly fails for an empty
path. (Fixed by Hirokazu Yamamoto; bpo-5913.)
- The mimelib module will now read the MIME database from the Windows registry when initializing. (Patch by Gabriel Genellina; bpo-4969.)

Port-Specific Changes: Mac OS X

- The path /Library/Python/2.7/site-packages is now appended to sys.path, in order to share added packages between the system installation and a user-installed copy of the same version. (Changed by Ronald Oussoren; bpo-4865.)

  Changed in version 2.7.13: As of 2.7.13, this change was removed. /Library/Python/2.7/site-packages, the site-packages directory used by the Apple-supplied system Python 2.7 is no longer appended to sys.path for user-installed Pythons such as from the python.org installers. As of macOS 10.12, Apple changed how the system site-packages directory is configured, which could cause installation of pip components, like setuptools, to fail. Packages installed for the system Python will no longer be shared with user-installed Pythons. (bpo-28440)

Port-Specific Changes: FreeBSD

- FreeBSD 7.1's SO_SETFIB constant, used with getsockopt() / setsockopt() to select an alternate routing table, is now available in the socket module. (Added by Kyle VanderBeek; bpo-8235.)
Other Changes and Fixes

- Two benchmark scripts, `iobench` and `ccbench`, were added to the `Tools` directory. `iobench` measures the speed of the built-in file I/O objects returned by `open()` while performing various operations, and `ccbench` is a concurrency benchmark that tries to measure computing throughput, thread switching latency, and IO processing bandwidth when performing several tasks using a varying number of threads.
- The `Tools/i18n/msgfmt.py` script now understands plural forms in `.po` files. (Fixed by Martin von Löwis; bpo-5464.)
- When importing a module from a `.pyc` or `.pyo` file with an existing `.py` counterpart, the `co_filename` attributes of the resulting code objects are overwritten when the original filename is obsolete. This can happen if the file has been renamed, moved, or is accessed through different paths. (Patch by Ziga Seilnacht and Jean-Paul Calderone; bpo-1180193.)
- The `regrtest.py` script now takes a `--randseed=` switch that takes an integer that will be used as the random seed for the `-r` option that executes tests in random order. The `-r` option also reports the seed that was used (Added by Collin Winter.)
- Another `regrtest.py` switch is `-j`, which takes an integer specifying how many tests run in parallel. This allows reducing the total runtime on multi-core machines. This option is compatible with several other options, including the `-R` switch which is known to produce long runtimes. (Added by Antoine Pitrou, bpo-6152.) This can also be used with a new `-F` switch that runs selected tests in a loop until they fail. (Added by Antoine Pitrou; bpo-7312.)
- When executed as a script, the `py_compile.py` module now accepts `'-f'` as an argument, which will read standard input for the list of filenames to be compiled. (Contributed by Piotr Ożarowski; bpo-8233.)
Porting to Python 2.7

This section lists previously described changes and other bug fixes that may require changes to your code:

- The `range()` function processes its arguments more consistently; it will now call `__int__()` on non-float, non-integer arguments that are supplied to it. (Fixed by Alexander Belopolsky; bpo-1533.)
- The string `format()` method changed the default precision used for floating-point and complex numbers from 6 decimal places to 12, which matches the precision used by `str()`. (Changed by Eric Smith; bpo-5920.)
- Because of an optimization for the `with` statement, the special methods `__enter__()` and `__exit__()` must belong to the object's type, and cannot be directly attached to the object's instance. This affects new-style classes (derived from `object`) and C extension types. (bpo-6101.)
- Due to a bug in Python 2.6, the `exc_value` parameter to `__exit__()` methods was often the string representation of the exception, not an instance. This was fixed in 2.7, so `exc_value` will be an instance as expected. (Fixed by Florent Xicluna; bpo-7853.)
- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise `AttributeError` as you would expect. Fixed by Benjamin Peterson; bpo-7604.)

In the standard library:

- Operations with `datetime` instances that resulted in a year falling outside the supported range didn’t always raise `OverflowError`. Such errors are now checked more carefully and will now raise the exception. (Reported by Mark Leander,
When using `Decimal` instances with a string's `format()` method, the default alignment was previously left-alignment. This has been changed to right-alignment, which might change the output of your programs. (Changed by Mark Dickinson; bpo-6857.)

Comparisons involving a signaling NaN value (or `sNAN`) now signal `InvalidOperation` instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; bpo-7279.)

The ElementTree library, `xml.etree`, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!– comment –>`). (Patch by Neil Muller; bpo-2746.)

The `readline()` method of `StringIO` objects now does nothing when a negative length is requested, as other file-like objects do. (bpo-7348).

The `syslog` module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of 'python'. (Changed by Sean Reifschneider; bpo-8451.)

The `tarfile` module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error...
level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; bpo-7357.)

- The `urlparse` module's `urlsplit()` now handles unknown URL schemes in a fashion compliant with RFC 3986: if the URL is of the form "<something>://...", the text before the :// is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

  ```python
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', '', '//host/filename?query', '', '')
  ```

  Python 2.7 (and Python 2.6.5) will return:

  ```python
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', 'host', '/filename?query', '', '')
  ```

  (Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

For C extensions:

- C extensions that use integer format codes with the `PyArg_Parse*` family of functions will now raise a `TypeError` exception instead of triggering a `DeprecationWarning` (bpo-5080).
- Use the new `PyOS_string_to_double()` function instead of the old `PyOS_ascii_strtod()` and `PyOS_ascii_atof()` functions, which are now deprecated.
For applications that embed Python:

- The `PySys_SetArgvEx()` function was added, letting applications close a security hole when the existing `PySys_SetArgv()` function was used. Check whether you're calling `PySys_SetArgv()` and carefully consider whether the application should be using `PySys_SetArgvEx()` with `updatepath` set to false.
New Features Added to Python 2.7 Maintenance Releases

New features may be added to Python 2.7 maintenance releases when the situation genuinely calls for it. Any such additions must go through the Python Enhancement Proposal process, and make a compelling case for why they can't be adequately addressed by either adding the new feature solely to Python 3, or else by publishing it on the Python Package Index.

In addition to the specific proposals listed below, there is a general exemption allowing new `-3` warnings to be added in any Python 2.7 maintenance release.

Two new environment variables for debug mode

In debug mode, the `[xxx refs]` statistic is not written by default, the `PYTHONSHOWREFCOUNT` environment variable now must also be set. (Contributed by Victor Stinner; bpo-31733.)

When Python is compiled with `COUNT_ALLOC` defined, allocation counts are no longer dumped by default anymore: the `PYTHONSHOWALLOC` environment variable must now also be set. Moreover, allocation counts are now dumped into stderr, rather than stdout. (Contributed by Victor Stinner; bpo-31692.)

*New in version 2.7.15.*

PEP 434: IDLE Enhancement Exception for All Branches
PEP 434 describes a general exemption for changes made to the IDLE development environment shipped along with Python. This exemption makes it possible for the IDLE developers to provide a more consistent user experience across all supported versions of Python 2 and 3.

For details of any IDLE changes, refer to the NEWS file for the specific release.

PEP 466: Network Security Enhancements for Python 2.7

PEP 466 describes a number of network security enhancement proposals that have been approved for inclusion in Python 2.7 maintenance releases, with the first of those changes appearing in the Python 2.7.7 release.

PEP 466 related features added in Python 2.7.7:

- `hmac.compare_digest()` was backported from Python 3 to make a timing attack resistant comparison operation available to Python 2 applications. (Contributed by Alex Gaynor; bpo-21306.)
- OpenSSL 1.0.1g was upgraded in the official Windows installers published on python.org. (Contributed by Zachary Ware; bpo-21462.)

PEP 466 related features added in Python 2.7.8:

- `hashlib.pbkdf2_hmac()` was backported from Python 3 to make a hashing algorithm suitable for secure password storage broadly available to Python 2 applications. (Contributed by Alex Gaynor; bpo-21304.)
- OpenSSL 1.0.1h was upgraded for the official Windows installers
published on python.org. (contributed by Zachary Ware in bpo-21671 for CVE-2014-0224)

**PEP 466** related features added in Python 2.7.9:

- Most of Python 3.4’s `ssl` module was backported. This means `ssl` now supports Server Name Indication, TLS1.x settings, access to the platform certificate store, the `SSLContext` class, and other features. (Contributed by Alex Gaynor and David Reid; bpo-21308.)

  Refer to the “Version added: 2.7.9” notes in the module documentation for specific details.

- `os.urandom()` was changed to cache a file descriptor to `/dev/urandom` instead of reopening `/dev/urandom` on every call. (Contributed by Alex Gaynor; bpo-21305.)

- `hashlib.algorithms_guaranteed` and `hashlib.algorithms_available` were backported from Python 3 to make it easier for Python 2 applications to select the strongest available hash algorithm. (Contributed by Alex Gaynor in bpo-21307)

**PEP 477: Backport ensurepip (PEP 453) to Python 2.7**

**PEP 477** approves the inclusion of the **PEP 453** ensurepip module and the improved documentation that was enabled by it in the Python 2.7 maintenance releases, appearing first in the Python 2.7.9 release.

**Bootstrapping pip By Default**
The new `ensurepip` module (defined in PEP 453) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations. The version of `pip` included with Python 2.7.9 is `pip 1.5.6`, and future 2.7.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pip`, `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies.

For CPython source builds on POSIX systems, the `make install` and `make altinstall` commands do not bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As discussed in the PEP, platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

**Documentation Changes**

As part of this change, the Installing Python Modules and Distributing
Python Modules sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as Installing Python Modules (Legacy version) and Distributing Python Modules (Legacy version).

**See also:**

**PEP 453** – Explicit bootstrapping of pip in Python installations

PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

**PEP 476:** Enabling certificate verification by default for stdlib http clients

**PEP 476** updated httplib and modules which use it, such as urllib2 and xmlrpclib, to now verify that the server presents a certificate which is signed by a Certificate Authority in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications. This change was made in the Python 2.7.9 release.

For applications which require the old previous behavior, they can pass an alternate context:

```python
import urllib2
import ssl
```
PEP 493: HTTPS verification migration tools for Python 2.7

PEP 493 provides additional migration tools to support a more incremental infrastructure upgrade process for environments containing applications and services relying on the historically permissive processing of server certificates when establishing client HTTPS connections. These additions were made in the Python 2.7.12 release.

These tools are intended for use in cases where affected applications and services can’t be modified to explicitly pass a more permissive SSL context when establishing the connection.

For applications and services which can’t be modified at all, the new PYTHONHTTPSVERIFY environment variable may be set to 0 to revert an entire Python process back to the default permissive behaviour of Python 2.7.8 and earlier.

For cases where the connection establishment code can’t be modified, but the overall application can be, the new ssl._https_verify_certificates() function can be used to
adjust the default behaviour at runtime.

**New make regen-all build target**

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new `make regen-all` command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see `Makefile.pre.in` for details.

(Contributed by Victor Stinner in [bpo-23404](https://bugs.python.org/issue23404).)

*New in version 2.7.14.*

**Removal of make touch build target**

The `make touch` build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new `make regen-all` target.

(Contributed by Victor Stinner in [bpo-23404](https://bugs.python.org/issue23404).)

*Changed in version 2.7.14.*
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Nick Coghlan, Philip Jenvey, Ryan Lovett, R. David Murray, Hugh Secker-Walker.
What’s New in Python 2.6

Author: A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.6, released on October 1 2008. The release schedule is described in PEP 361.

The major theme of Python 2.6 is preparing the migration path to Python 3.0, a major redesign of the language. Whenever possible, Python 2.6 incorporates new features and syntax from 3.0 while remaining compatible with existing code by not removing older features or syntax. When it's not possible to do that, Python 2.6 tries to do what it can, adding compatibility functions in a future_builtins module and a -3 switch to warn about usages that will become unsupported in 3.0.

Some significant new packages have been added to the standard library, such as the multiprocessing and json modules, but there aren’t many new features that aren’t related to Python 3.0 in some way.

Python 2.6 also sees a number of improvements and bugfixes throughout the source. A search through the change logs finds there were 259 patches applied and 612 bugs fixed between Python 2.5 and 2.6. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.6. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature. Whenever possible, “What’s New in Python” links to the bug/patch item for each change.
The development cycle for Python versions 2.6 and 3.0 was synchronized, with the alpha and beta releases for both versions being made on the same days. The development of 3.0 has influenced many features in 2.6.

Python 3.0 is a far-ranging redesign of Python that breaks compatibility with the 2.x series. This means that existing Python code will need some conversion in order to run on Python 3.0. However, not all the changes in 3.0 necessarily break compatibility. In cases where new features won’t cause existing code to break, they’ve been backported to 2.6 and are described in this document in the appropriate place. Some of the 3.0-derived features are:

- A `__complex__() method for converting objects to a complex number.
- Alternate syntax for catching exceptions: `except(TypeError as exc`
- The addition of `functools.reduce()` as a synonym for the built-in `reduce()` function.

Python 3.0 adds several new built-in functions and changes the semantics of some existing builtins. Functions that are new in 3.0 such as `bin()` have simply been added to Python 2.6, but existing builtins haven’t been changed; instead, the `future_builtins` module has versions with the new 3.0 semantics. Code written to be compatible with 3.0 can do `from future_builtins import hex, map as necessary.`

A new command-line switch, `-3`, enables warnings about features that will be removed in Python 3.0. You can run code with this switch to see
how much work will be necessary to port code to 3.0. The value of this switch is available to Python code as the boolean variable `sys.py3kwarning`, and to C extension code as `Py_Py3kWarningFlag`.

**See also:** The 3xxx series of PEPs, which contains proposals for Python 3.0. [PEP 3000](https://www.python.org/dev/peps/pep-0300/) describes the development process for Python 3.0. Start with [PEP 3100](https://www.python.org/dev/peps/pep-03100/) that describes the general goals for Python 3.0, and then explore the higher-numbered PEPS that propose specific features.
Changes to the Development Process

While 2.6 was being developed, the Python development process underwent two significant changes: we switched from SourceForge's issue tracker to a customized Roundup installation, and the documentation was converted from LaTeX to reStructuredText.

New Issue Tracker: Roundup

For a long time, the Python developers had been growing increasingly annoyed by SourceForge's bug tracker. SourceForge's hosted solution doesn’t permit much customization; for example, it wasn’t possible to customize the life cycle of issues.

The infrastructure committee of the Python Software Foundation therefore posted a call for issue trackers, asking volunteers to set up different products and import some of the bugs and patches from SourceForge. Four different trackers were examined: Jira, Launchpad, Roundup, and Trac. The committee eventually settled on Jira and Roundup as the two candidates. Jira is a commercial product that offers no-cost hosted instances to free-software projects; Roundup is an open-source project that requires volunteers to administer it and a server to host it.

After posting a call for volunteers, a new Roundup installation was set up at https://bugs.python.org. One installation of Roundup can host multiple trackers, and this server now also hosts issue trackers for Jython and for the Python web site. It will surely find other uses in the future. Where possible, this edition of “What’s New in Python” links to the bug/patch item for each change.
Hosting of the Python bug tracker is kindly provided by Upfront Systems of Stellenbosch, South Africa. Martin von Löwis put a lot of effort into importing existing bugs and patches from SourceForge; his scripts for this import operation are at http://svn.python.org/view/tracker/importer/ and may be useful to other projects wishing to move from SourceForge to Roundup.

See also:

https://bugs.python.org
The Python bug tracker.

http://bugs.jython.org:
The Jython bug tracker.

http://roundup.sourceforge.net/
Roundup downloads and documentation.

http://svn.python.org/view/tracker/importer/
Martin von Löwis's conversion scripts.

New Documentation Format: reStructuredText
Using Sphinx

The Python documentation was written using LaTeX since the project started around 1989. In the 1980s and early 1990s, most documentation was printed out for later study, not viewed online. LaTeX was widely used because it provided attractive printed output while remaining straightforward to write once the basic rules of the markup were learned.

Today LaTeX is still used for writing publications destined for printing, but the landscape for programming tools has shifted. We no longer print out reams of documentation; instead, we browse through it online
and HTML has become the most important format to support. Unfortunately, converting LaTeX to HTML is fairly complicated and Fred L. Drake Jr., the long-time Python documentation editor, spent a lot of time maintaining the conversion process. Occasionally people would suggest converting the documentation into SGML and later XML, but performing a good conversion is a major task and no one ever committed the time required to finish the job.

During the 2.6 development cycle, Georg Brandl put a lot of effort into building a new toolchain for processing the documentation. The resulting package is called Sphinx, and is available from http://sphinx-doc.org/.

Sphinx concentrates on HTML output, producing attractively styled and modern HTML; printed output is still supported through conversion to LaTeX. The input format is reStructuredText, a markup syntax supporting custom extensions and directives that is commonly used in the Python community.

Sphinx is a standalone package that can be used for writing, and almost two dozen other projects (listed on the Sphinx web site) have adopted Sphinx as their documentation tool.

See also:

**Documenting Python**
Describes how to write for Python’s documentation.

**Sphinx**
Documentation and code for the Sphinx toolchain.

**Docutils**
The underlying reStructuredText parser and toolset.
PEP 343: The ‘with’ statement

The previous version, Python 2.5, added the ‘with’ statement as an optional feature, to be enabled by a `from __future__ import with_statement` directive. In 2.6 the statement no longer needs to be specially enabled; this means that `with` is now always a keyword. The rest of this section is a copy of the corresponding section from the “What’s New in Python 2.5” document; if you’re familiar with the ‘with’ statement from Python 2.5, you can skip this section.

The ‘with’ statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I’ll discuss the statement as it will commonly be used. In the next section, I’ll examine the implementation details and show how to write objects for use with this statement.

The ‘with’ statement is a control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods).

The object’s `__enter__()` is called before `with-block` is executed and therefore can run set-up code. It also may return a value that is bound to the name `variable`, if given. (Note carefully that `variable` is not assigned the result of `expression`.)

After execution of the `with-block` is finished, the object’s `__exit__()`
method is called, even if the block raised an exception, and can therefore run clean-up code.

Some standard Python objects now support the context management protocol and can be used with the ‘with’ statement. File objects are one example:

```python
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more processing code ...
```

After this statement has executed, the file object in f will have been automatically closed, even if the for loop raised an exception part-way through the block.

**Note:** In this case, f is the same object created by open(), because file.__enter__() returns self.

The `threading` module’s locks and condition variables also support the ‘with’ statement:

```python
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for
computations:

```python
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print(v.sqrt())

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16
    # The original context is restored on exiting the
    print(v.sqrt())
```

Writing Context Managers

Under the hood, the `with` statement is fairly complicated. Most people will only use `with` in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager’s `__enter__()` method is called. The value returned is assigned to `VAR`. If no `as VAR` clause is present, the value is simply discarded.
- The code in `BLOCK` is executed.
- If `BLOCK` raises an exception, the context manager’s `__exit__()` method is called with three arguments, the exception details (`type`, `value`, `traceback`), the same values
returned by `sys.exc_info()`, which can also be `None` if no exception occurred). The method's return value controls whether an exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You'll only rarely want to suppress the exception, because if you do the author of the code containing the `with` statement will never realize anything went wrong.

- If `BLOCK` didn't raise an exception, the `__exit__()` method is still called, but `type`, `value`, and `traceback` are all `None`.

Let's think through an example. I won't present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction. Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let's assume there's an object representing a database connection. Our goal will be to let the user write code like this:

```python
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there's an exception. Here's the basic interface for `DatabaseConnection` that I'll assume:
The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add as `cursor` to their `with` statement to bind the cursor to a variable name.

The `__exit__()` method is the most complicated because it's where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of `None`. `None` is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.
... 

def __exit__(self, type, value, tb):
    if tb is None:
        # No exception, so commit
        self.commit()
    else:
        # Exception occurred, so rollback.
        self.rollback()
        # return False

The contextlib module

The contextlib module provides some functions and a decorator that are useful when writing objects for use with the 'with' statement.

The decorator is called contextmanager(), and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the yield will be executed as the __enter__() method, and the value yielded will be the method's return value that will get bound to the variable in the 'with' statement's as clause, if any. The code after the yield will be executed in the __exit__() method. Any exception raised in the block will be raised by the yield statement.

Using this decorator, our database example from the previous section could be written as:

```python
from contextlib import contextmanager

@contextmanager
def db_transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
```
except:
    connection.rollback()
    raise
else:
    connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...

The `contextlib` module also has a `nested(mgr1, mgr2, ...)` function that combines a number of context managers so you don't need to write nested `with` statements. In this example, the single `with` statement both starts a database transaction and acquires a thread lock:

```
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, lock):
    ...
```

Finally, the `closing()` function returns its argument so that it can be bound to a variable, and calls the argument's `.close()` method at the end of the block.

```
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com'))
    for line in f:
        sys.stdout.write(line)
```

See also:
PEP 343 - The “with” statement

PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a ‘with’ statement, which can be helpful in learning how the statement works.

The documentation for the contextlib module.
PEP 366: Explicit Relative Imports From a Main Module

Python’s -m switch allows running a module as a script. When you ran a module that was located inside a package, relative imports didn’t work correctly.

The fix for Python 2.6 adds a __package__ attribute to modules. When this attribute is present, relative imports will be relative to the value of this attribute instead of the __name__ attribute.

PEP 302-style importers can then set __package__ as necessary. The runpy module that implements the -m switch now does this, so relative imports will now work correctly in scripts running from inside a package.
PEP 370: Per-user site-packages Directory

When you run Python, the module search path `sys.path` usually includes a directory whose path ends in "site-packages". This directory is intended to hold locally-installed packages available to all users using a machine or a particular site installation.

Python 2.6 introduces a convention for user-specific site directories. The directory varies depending on the platform:

- Unix and Mac OS X: `~/.local/`
- Windows: `%APPDATA%/Python`

Within this directory, there will be version-specific subdirectories, such as `lib/python2.6/site-packages` on Unix/Mac OS and `Python26/site-packages` on Windows.

If you don’t like the default directory, it can be overridden by an environment variable. `PYTHONUSERBASE` sets the root directory used for all Python versions supporting this feature. On Windows, the directory for application-specific data can be changed by setting the `APPDATA` environment variable. You can also modify the `site.py` file for your Python installation.

The feature can be disabled entirely by running Python with the `-s` option or setting the `PYTHONNOUSERSITE` environment variable.

See also:

- PEP 370 - Per-user site-packages Directory
  PEP written and implemented by Christian Heimes.
PEP 371: The multiprocessing Package

The new `multiprocessing` package lets Python programs create new processes that will perform a computation and return a result to the parent. The parent and child processes can communicate using queues and pipes, synchronize their operations using locks and semaphores, and can share simple arrays of data.

The `multiprocessing` module started out as an exact emulation of the `threading` module using processes instead of threads. That goal was discarded along the path to Python 2.6, but the general approach of the module is still similar. The fundamental class is the `Process`, which is passed a callable object and a collection of arguments. The `start()` method sets the callable running in a subprocess, after which you can call the `is_alive()` method to check whether the subprocess is still running and the `join()` method to wait for the process to exit.

Here's a simple example where the subprocess will calculate a factorial. The function doing the calculation is written strangely so that it takes significantly longer when the input argument is a multiple of 4.

```python
import time
from multiprocessing import Process, Queue

def factorial(queue, N):
    """Compute a factorial.""
    # If N is a multiple of 4, this function will take
    if (N % 4) == 0:
        time.sleep(.05 * N/4)
```
A **Queue** is used to communicate the result of the factorial. The **Queue** object is stored in a global variable. The child process will use the value of the variable when the child was created; because it's a **Queue**, parent and child can use the object to communicate. (If the parent were to change the value of the global variable, the child's value would be unaffected, and vice versa.)

Two other classes, **Pool** and **Manager**, provide higher-level interfaces. **Pool** will create a fixed number of worker processes, and requests can then be distributed to the workers by calling **apply()** or **apply_async()** to add a single request, and **map()** or **map_async()** to add a number of requests. The following code uses a **Pool** to spread requests across 5 worker processes and retrieve a
list of results:

```python
from multiprocessing import Pool

def factorial(N, dictionary):
    "Compute a factorial."
    ...
p = Pool(5)
result = p.map(factorial, range(1, 1000, 10))
for v in result:
    print v
```

This produces the following output:

```
1
39916800
51090942171709440000
8222838654177922817725562880000000
33452526613163807108170062053440751665152000000000
...
```

The other high-level interface, the `Manager` class, creates a separate server process that can hold master copies of Python data structures. Other processes can then access and modify these data structures using proxy objects. The following example creates a shared dictionary by calling the `dict()` method; the worker processes then insert values into the dictionary. (Locking is not done for you automatically, which doesn’t matter in this example. Manager’s methods also include `Lock()`, `RLock()`, and `Semaphore()` to create shared locks.)

```python
import time
from multiprocessing import Pool, Manager
```
def factorial(N, dictionary):
    "Compute a factorial."
    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Store result in dictionary
    dictionary[N] = fact

if __name__ == '__main__':
    p = Pool(5)
    mgr = Manager()
    d = mgr.dict()  # Create shared dictionary

    # Run tasks using the pool
    for N in range(1, 1000, 10):
        p.apply_async(factorial, (N, d))

    # Mark pool as closed -- no more tasks can be added
    p.close()

    # Wait for tasks to exit
    p.join()

    # Output results
    for k, v in sorted(d.items()):
        print k, v

This will produce the output:

1 1
11 39916800
21 51090942171709440000
31 82228386541779228177255628800000000
41 33452526613163807108170062053440751665152000000000
See also: The documentation for the `multiprocessing` module.

**PEP 371 - Addition of the multiprocessing package**

PEP written by Jesse Noller and Richard Oudkerk; implemented by Richard Oudkerk and Jesse Noller.
PEP 3101: Advanced String Formatting

In Python 3.0, the % operator is supplemented by a more powerful string formatting method, `format()`. Support for the `str.format()` method has been backported to Python 2.6.

In 2.6, both 8-bit and Unicode strings have a `.format()` method that treats the string as a template and takes the arguments to be formatted. The formatting template uses curly brackets ({, }) as special characters:

```python
>>> # Substitute positional argument 0 into the string
>>> "User ID: {0}".format("root")
'User ID: root'

>>> # Use the named keyword arguments
>>> "User ID: {uid} Last seen: {last_login}".format(uid="root",
... uid="root",
... last_login = "5 Mar 2008 07:20")
'User ID: root Last seen: 5 Mar 2008 07:20'
```

Curly brackets can be escaped by doubling them:

```python
>>> "Empty dict: {{}}".format()
"Empty dict: {}
```

Field names can be integers indicating positional arguments, such as `{0}`, `{1}`, etc. or names of keyword arguments. You can also supply compound field names that read attributes or access dictionary keys:

```python
>>> import sys
>>> print 'Platform: {0.platform}\nPython version: {1}'
Platform: darwin
Python version: 2.6a1+ (trunk:61261M, Mar 5 2008, 2
>>> import mimetypes
>>> 'Content-type: {0[.mp4]}'.format(mimetypes.types_map['.mp4'])
'Content-type: video/mp4'

Note that when using dictionary-style notation such as [.mp4], you don’t need to put any quotation marks around the string; it will look up the value using .mp4 as the key. Strings beginning with a number will be converted to an integer. You can’t write more complicated expressions inside a format string.

So far we’ve shown how to specify which field to substitute into the resulting string. The precise formatting used is also controllable by adding a colon followed by a format specifier. For example:

```python
>>> # Field 0: left justify, pad to 15 characters
>>> # Field 1: right justify, pad to 6 characters
>>> fmt = '{0:15} ${1:>6}'
>>> fmt.format('Registration', 35)
'Registration $ 35'
>>> fmt.format('Tutorial', 50)
'Tutorial $ 50'
>>> fmt.format('Banquet', 125)
'Banquet $ 125'
```

Format specifiers can reference other fields through nesting:

```python
>>> fmt = '{0:{1}}'
>>> width = 15
>>> fmt.format('Invoice #1234', width)
'Invoice #1234'
>>> width = 35
>>> fmt.format('Invoice #1234', width)
'Invoice #1234'
```
The alignment of a field within the desired width can be specified:

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; (default)</td>
<td>Left-align</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-align</td>
</tr>
<tr>
<td>^</td>
<td>Center</td>
</tr>
<tr>
<td>=</td>
<td>(For numeric types only) Pad after the sign.</td>
</tr>
</tbody>
</table>

Format specifiers can also include a presentation type, which controls how the value is formatted. For example, floating-point numbers can be formatted as a general number or in exponential notation:

```python
>>> '{0:g}'.format(3.75)
'3.75'
>>> '{0:e}'.format(3.75)
'3.750000e+00'
```

A variety of presentation types are available. Consult the 2.6 documentation for a complete list; here's a sample:

<p>| b | Binary. Outputs the number in base 2. |
|   |                      | |
| c | Character. Converts the integer to the corresponding Unicode character before printing. |
| d | Decimal Integer. Outputs the number in base 10. |
| o | Octal format. Outputs the number in base 8. |
| x | Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9. |
| e | Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. |</p>
<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>g</strong></td>
<td>General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to ‘e’ exponent notation.</td>
</tr>
<tr>
<td><strong>n</strong></td>
<td>Number. This is the same as ‘g’ (for floats) or ‘d’ (for integers), except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td><strong>%</strong></td>
<td>Percentage. Multiplies the number by 100 and displays in fixed (‘f’) format, followed by a percent sign.</td>
</tr>
</tbody>
</table>

Classes and types can define a `__format__()` method to control how they're formatted. It receives a single argument, the format specifier:

```python
def __format__(self, format_spec):
    if isinstance(format_spec, unicode):
        return unicode(str(self))
    else:
        return str(self)
```

There's also a `format()` builtin that will format a single value. It calls the type's `__format__()` method with the provided specifier:

```python
>>> format(75.6564, '.2f')
'75.66'
```

**See also:**

**Format String Syntax**  
The reference documentation for format fields.

**PEP 3101 - Advanced String Formatting**  
PEP written by Talin. Implemented by Eric Smith.
PEP 3105: print As a Function

The `print` statement becomes the `print()` function in Python 3.0. Making `print()` a function makes it possible to replace the function by doing `def print(...)` or importing a new function from somewhere else.

Python 2.6 has a `__future__` import that removes `print` as language syntax, letting you use the functional form instead. For example:

```python
>>> from __future__ import print_function
>>> print('# of entries', len(dictionary), file=sys.
```

The signature of the new function is:

```python
def print(*args, sep=' ', end='
', file=None)
```

The parameters are:

- `args`: positional arguments whose values will be printed out.
- `sep`: the separator, which will be printed between arguments.
- `end`: the ending text, which will be printed after all of the arguments have been output.
- `file`: the file object to which the output will be sent.

See also:

**PEP 3105 - Make print a function**

PEP written by Georg Brandl.
PEP 3110: Exception-Handling Changes

One error that Python programmers occasionally make is writing the following code:

```python
try:
  ...
except TypeError, ValueError:  # Wrong!
  ...
```

The author is probably trying to catch both `TypeError` and `ValueError` exceptions, but this code actually does something different: it will catch `TypeError` and bind the resulting exception object to the local name "ValueError". The `ValueError` exception will not be caught at all. The correct code specifies a tuple of exceptions:

```python
try:
  ...
except (TypeError, ValueError):
  ...
```

This error happens because the use of the comma here is ambiguous: does it indicate two different nodes in the parse tree, or a single node that's a tuple?

Python 3.0 makes this unambiguous by replacing the comma with the word “as”. To catch an exception and store the exception object in the variable `exc`, you must write:

```python
try:
  ...
except TypeError as exc:
```

Python 3.0 will only support the use of “as”, and therefore interprets the first example as catching two different exceptions. Python 2.6 supports both the comma and “as”, so existing code will continue to work. We therefore suggest using “as” when writing new Python code that will only be executed with 2.6.

See also:

**PEP 3110 - Catching Exceptions in Python 3000**
PEP written and implemented by Collin Winter.
PEP 3112: Byte Literals

Python 3.0 adopts Unicode as the language's fundamental string type and denotes 8-bit literals differently, either as \b'string\' or using a bytes constructor. For future compatibility, Python 2.6 adds bytes as a synonym for the str type, and it also supports the b'' notation.

The 2.6 str differs from 3.0's bytes type in various ways; most notably, the constructor is completely different. In 3.0, \texttt{bytes([65, 66, 67])} is 3 elements long, containing the bytes representing ABC; in 2.6, \texttt{bytes([65, 66, 67])} returns the 12-byte string representing the str() of the list.

The primary use of bytes in 2.6 will be to write tests of object type such as \texttt{isinstance(x, bytes)}. This will help the 2to3 converter, which can't tell whether 2.x code intends strings to contain either characters or 8-bit bytes; you can now use either bytes or str to represent your intention exactly, and the resulting code will also be correct in Python 3.0.

There's also a __future__ import that causes all string literals to become Unicode strings. This means that \\u escape sequences can be used to include Unicode characters:

```python
from __future__ import unicode_literals
s = ('\u751f\u3080\u304e\u3000\u751f\u3054'\
     '\u3081\u3000\u751f\u305f\u307e\u3054'
)
print len(s) # 12 Unicode characters
```

At the C level, Python 3.0 will rename the existing 8-bit string type,
called PyStringObject in Python 2.x, to `PyBytesObject`. Python 2.6 uses `#define` to support using the names `PyBytesObject()`, `PyBytes_Check()`, `PyBytes_FromStringAndSize()`, and all the other functions and macros used with strings.

Instances of the `bytes` type are immutable just as strings are. A new `bytearray` type stores a mutable sequence of bytes:

```python
>>> bytearray([65, 66, 67])
bytearray(b'ABC')
>>> b = bytearray(u'\u21ef\u3244', 'utf-8')
>>> b
bytearray(b'\xe2\x87\xa4\xe3\x89\x84')
>>> b[0] = '\xe3'
>>> b
bytearray(b'\xe3\x87\xa4\xe3\x89\x84')
>>> unicode(str(b), 'utf-8')
u'\u31ef \u3244'
```

Byte arrays support most of the methods of string types, such as `startswith()`/`endswith()`, `find()`/`rfind()`, and some of the methods of lists, such as `append()`, `pop()`, and `reverse()`.

```python
>>> b = bytearray('ABC')
>>> b.append('d')
>>> b.append(ord('e'))
>>> b
bytearray(b'ABCde')
```

There's also a corresponding C API, with `PyByteArray_FromObject()`, `PyByteArray_FromStringAndSize()`, and various other functions.
See also:

**PEP 3112 - Bytes literals in Python 3000**
PEP written by Jason Orendorff; backported to 2.6 by Christian Heimes.
PEP 3116: New I/O Library

Python’s built-in file objects support a number of methods, but file-like objects don’t necessarily support all of them. Objects that imitate files usually support read() and write(), but they may not support readline(), for example. Python 3.0 introduces a layered I/O library in the io module that separates buffering and text-handling features from the fundamental read and write operations.

There are three levels of abstract base classes provided by the io module:

- **RawIOBase** defines raw I/O operations: read(), readinto(), write(), seek(), tell(), truncate(), and close(). Most of the methods of this class will often map to a single system call. There are also readable(), writable(), and seekable() methods for determining what operations a given object will allow.

  Python 3.0 has concrete implementations of this class for files and sockets, but Python 2.6 hasn’t restructured its file and socket objects in this way.

- **BufferedIOBase** is an abstract base class that buffers data in memory to reduce the number of system calls used, making I/O processing more efficient. It supports all of the methods of RawIOBase, and adds a raw attribute holding the underlying raw object.

  There are five concrete classes implementing this ABC. BufferedWriter and BufferedReader are for objects that support write-only or read-only usage that have a seek() method
for random access. BufferedRandom objects support read and write access upon the same underlying stream, and BufferedRWPair is for objects such as TTYs that have both read and write operations acting upon unconnected streams of data. The BytesIO class supports reading, writing, and seeking over an in-memory buffer.

- **TextIOBase**: Provides functions for reading and writing strings (remember, strings will be Unicode in Python 3.0), and supporting *universal newlines*. TextIOBase defines the `readline()` method and supports iteration upon objects.

There are two concrete implementations. TextIOWrapper wraps a buffered I/O object, supporting all of the methods for text I/O and adding a buffer attribute for access to the underlying object. StringIO simply buffers everything in memory without ever writing anything to disk.

(In Python 2.6, `io.StringIO` is implemented in pure Python, so it’s pretty slow. You should therefore stick with the existing StringIO module or cStringIO for now. At some point Python 3.0’s `io` module will be rewritten into C for speed, and perhaps the C implementation will be backported to the 2.x releases.)

In Python 2.6, the underlying implementations haven’t been restructured to build on top of the `io` module’s classes. The module is being provided to make it easier to write code that’s forward-compatible with 3.0, and to save developers the effort of writing their own implementations of buffering and text I/O.

**See also:**
PEP 3116 - New I/O

PEP 3118: Revised Buffer Protocol

The buffer protocol is a C-level API that lets Python types exchange pointers into their internal representations. A memory-mapped file can be viewed as a buffer of characters, for example, and this lets another module such as `re` treat memory-mapped files as a string of characters to be searched.

The primary users of the buffer protocol are numeric-processing packages such as NumPy, which expose the internal representation of arrays so that callers can write data directly into an array instead of going through a slower API. This PEP updates the buffer protocol in light of experience from NumPy development, adding a number of new features such as indicating the shape of an array or locking a memory region.

The most important new C API function is `PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)`, which takes an object and a set of flags, and fills in the `Py_buffer` structure with information about the object's memory representation. Objects can use this operation to lock memory in place while an external caller could be modifying the contents, so there's a corresponding `PyBuffer_Release(Py_buffer *view)` to indicate that the external caller is done.

The `flags` argument to `PyObject_GetBuffer()` specifies constraints upon the memory returned. Some examples are:

- `PyBUF_WRITABLE` indicates that the memory must be writable.
- `PyBUF_LOCK` requests a read-only or exclusive lock on the memory.
- PyBUF_C_CONTIGUOUS and PyBUF_F_CONTIGUOUS requests a C-contiguous (last dimension varies the fastest) or Fortran-contiguous (first dimension varies the fastest) array layout.

Two new argument codes for `PyArg_ParseTuple()`, `s*` and `z*`, return locked buffer objects for a parameter.

See also:

**PEP 3118 - Revising the buffer protocol**
PEP written by Travis Oliphant and Carl Banks; implemented by Travis Oliphant.
PEP 3119: Abstract Base Classes

Some object-oriented languages such as Java support interfaces, declaring that a class has a given set of methods or supports a given access protocol. Abstract Base Classes (or ABCs) are an equivalent feature for Python. The ABC support consists of an abc module containing a metaclass called ABCMeta, special handling of this metaclass by the isinstance() and issubclass() builtins, and a collection of basic ABCs that the Python developers think will be widely useful. Future versions of Python will probably add more ABCs.

Let's say you have a particular class and wish to know whether it supports dictionary-style access. The phrase “dictionary-style” is vague, however. It probably means that accessing items with \texttt{obj[1]} works. Does it imply that setting items with \texttt{obj[2] = value} works? Or that the object will have \texttt{keys()}, \texttt{values()}, and \texttt{items()} methods? What about the iterative variants such as \texttt{iterkeys()}? \texttt{copy()} and \texttt{update()}? Iterating over the object with \texttt{iter()}?

The Python 2.6 collections module includes a number of different ABCs that represent these distinctions. Iterable indicates that a class defines \texttt{__iter__()}, and Container means the class defines a \texttt{__contains__()} method and therefore supports \texttt{x in y} expressions. The basic dictionary interface of getting items, setting items, and \texttt{keys()}, \texttt{values()}, and \texttt{items()}, is defined by the MutableMapping ABC.

You can derive your own classes from a particular ABC to indicate they support that ABC's interface:

\begin{verbatim}
import collections
\end{verbatim}
class Storage(collections.MutableMapping):
...

Alternatively, you could write the class without deriving from the desired ABC and instead register the class by calling the ABC's register() method:

import collections
class Storage:
...
collections.MutableMapping.register(Storage)

For classes that you write, deriving from the ABC is probably clearer. The register() method is useful when you've written a new ABC that can describe an existing type or class, or if you want to declare that some third-party class implements an ABC. For example, if you defined a PrintableType ABC, it's legal to do:

# Register Python's types
PrintableType.register(int)
PrintableType.register(float)
PrintableType.register(str)

Classes should obey the semantics specified by an ABC, but Python can't check this; it's up to the class author to understand the ABC's requirements and to implement the code accordingly.

To check whether an object supports a particular interface, you can now write:

def func(d):
if not isinstance(d, collections.MutableMapping):
    raise ValueError("Mapping object expected, got {}")

Don’t feel that you must now begin writing lots of checks as in the above example. Python has a strong tradition of duck-typing, where explicit type-checking is never done and code simply calls methods on an object, trusting that those methods will be there and raising an exception if they aren’t. Be judicious in checking for ABCs and only do it where it’s absolutely necessary.

You can write your own ABCs by using `abc.ABCMeta` as the metaclass in a class definition:

```python
from abc import ABCMeta, abstractmethod

class Drawable:
    __metaclass__ = ABCMeta

    @abstractmethod
def draw(self, x, y, scale=1.0):
        pass

    def draw_doubled(self, x, y):
        self.draw(x, y, scale=2.0)

class Square(Drawable):
    def draw(self, x, y, scale):
        ...
```

In the Drawable ABC above, the `draw_doubled()` method renders the object at twice its size and can be implemented in terms of other methods described in Drawable. Classes implementing this ABC therefore don’t need to provide their own implementation of
draw_doubled(), though they can do so. An implementation of draw() is necessary, though; the ABC can't provide a useful generic implementation.

You can apply the @abstractmethod decorator to methods such as draw() that must be implemented; Python will then raise an exception for classes that don't define the method. Note that the exception is only raised when you actually try to create an instance of a subclass lacking the method:

```python
>>> class Circle(Drawable):
...     pass
...
>>> c = Circle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Circle with abstract methods draw
```

Abstract data attributes can be declared using the @abstractproperty decorator:

```python
from abc import abstractproperty
...

@abstractproperty
def readonly(self):
    return self._x
```

Subclasses must then define a readonly() property.

See also:

**PEP 3119** - Introducing Abstract Base Classes
PEP written by Guido van Rossum and Talin. Implemented by Guido van Rossum. Backported to 2.6 by Benjamin Aranguren, with Alex Martelli.
PEP 3127: Integer Literal Support and Syntax

Python 3.0 changes the syntax for octal (base-8) integer literals, prefixing them with “0o” or “0O” instead of a leading zero, and adds support for binary (base-2) integer literals, signalled by a “0b” or “0B” prefix.

Python 2.6 doesn’t drop support for a leading 0 signalling an octal number, but it does add support for “0o” and “0b”:

```
>>> 0o21, 2*8 + 1
(17, 17)
>>> 0b10111
47
```

The `oct()` builtin still returns numbers prefixed with a leading zero, and a new `bin()` builtin returns the binary representation for a number:

```
>>> oct(42)
'052'
>>> future_builtins.oct(42)
'0o52'
>>> bin(173)
'0b10101101'
```

The `int()` and `long()` builtins will now accept the “0o” and “0b” prefixes when base-8 or base-2 are requested, or when the `base` argument is zero (signalling that the base used should be determined from the string):
>>> int ('0o52', 0)
42
>>> int('1101', 2)
13
>>> int('0b1101', 2)
13
>>> int('0b1101', 0)
13

See also:

**PEP 3127 - Integer Literal Support and Syntax**

PEP written by Patrick Maupin; backported to 2.6 by Eric Smith.
PEP 3129: Class Decorators

Decorators have been extended from functions to classes. It’s now legal to write:

```python
@foo
@bar
class A:
    pass
```

This is equivalent to:

```python
class A:
    pass
A = foo(bar(A))
```

See also:

**PEP 3129 - Class Decorators**

PEP written by Collin Winter.
PEP 3141: A Type Hierarchy for Numbers

Python 3.0 adds several abstract base classes for numeric types inspired by Scheme's numeric tower. These classes were backported to 2.6 as the numbers module.

The most general ABC is Number. It defines no operations at all, and only exists to allow checking if an object is a number by doing isinstance(obj, Number).

Complex is a subclass of Number. Complex numbers can undergo the basic operations of addition, subtraction, multiplication, division, and exponentiation, and you can retrieve the real and imaginary parts and obtain a number's conjugate. Python's built-in complex type is an implementation of Complex.

Real further derives from Complex, and adds operations that only work on real numbers: floor(), trunc(), rounding, taking the remainder mod N, floor division, and comparisons.

Rational numbers derive from Real, have numerator and denominator properties, and can be converted to floats. Python 2.6 adds a simple rational-number class, Fraction, in the fractions module. (It's called Fraction instead of Rational to avoid a name clash with numbers.Rational.)

Integral numbers derive from Rational, and can be shifted left and right with << and>>, combined using bitwise operations such as & and |, and can be used as array indexes and slice boundaries.

In Python 3.0, the PEP slightly redefines the existing builtins
round(), math.floor(), math.ceil(), and adds a new one, math.trunc(), that's been backported to Python 2.6. math.trunc() rounds toward zero, returning the closest Integral that's between the function's argument and zero.

See also:

PEP 3141 - A Type Hierarchy for Numbers
PEP written by Jeffrey Yasskin.

Scheme's numerical tower, from the Guile manual.

Scheme's number datatypes from the R5RS Scheme specification.

The fractions Module

To fill out the hierarchy of numeric types, the fractions module provides a rational-number class. Rational numbers store their values as a numerator and denominator forming a fraction, and can exactly represent numbers such as 2/3 that floating-point numbers can only approximate.

The Fraction constructor takes two Integral values that will be the numerator and denominator of the resulting fraction.

```python
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> float(a), float(b)
(0.6666666666666666, 0.40000000000000002)
>>> a+b
Fraction(16, 15)
>>> a/b
```
For converting floating-point numbers to rationals, the float type now has an `as_integer_ratio()` method that returns the numerator and denominator for a fraction that evaluates to the same floating-point value:

```python
>>> (2.5).as_integer_ratio()
(5, 2)
>>> (3.1415).as_integer_ratio()
(7074029114692207L, 2251799813685248L)
>>> (1./3).as_integer_ratio()
(6004799503160661L, 18014398509481984L)
```

Note that values that can only be approximated by floating-point numbers, such as `1./3`, are not simplified to the number being approximated; the fraction attempts to match the floating-point value exactly.

The `fractions` module is based upon an implementation by Sjoerd Mullender that was in Python's `Demo/classes/` directory for a long time. This implementation was significantly updated by Jeffrey Yasskin.
Other Language Changes

Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory or zip archive is automatically inserted as the first entry in `sys.path`. (Suggestion and initial patch by Andy Chu, subsequently revised by Phillip J. Eby and Nick Coghlan; bpo-1739468.)

- The `hasattr()` function was catching and ignoring all errors, under the assumption that they meant a `__getattr__()` method was failing somehow and the return value of `hasattr()` would therefore be `False`. This logic shouldn't be applied to `KeyboardInterrupt` and `SystemExit`, however; Python 2.6 will no longer discard such exceptions when `hasattr()` encounters them. (Fixed by Benjamin Peterson; bpo-2196.)

- When calling a function using the `**` syntax to provide keyword arguments, you are no longer required to use a Python dictionary; any mapping will now work:

```python
>>> def f(**kw):
...     print sorted(kw)
... 
>>> ud=UserDict.UserDict()
>>> ud['a'] = 1
>>> ud['b'] = 'string'
>>> f(**ud)
['a', 'b']
```
It's also become legal to provide keyword arguments after a *args argument to a function call.

```python
>>> def f(*args, **kw):
...     print args, kw
...
>>> f(1,2,3, *(4,5,6), keyword=13)
(1, 2, 3, 4, 5, 6) {'keyword': 13}
```

Previously this would have been a syntax error. (Contributed by Amaury Forgeot d'Arc; bpo-3473.)

- A new builtin, `next(iterator, [default])` returns the next item from the specified iterator. If the `default` argument is supplied, it will be returned if `iterator` has been exhausted; otherwise, the `StopIteration` exception will be raised. (Backported in bpo-2719.)

- Tuples now have `index()` and `count()` methods matching the list type's `index()` and `count()` methods:

```python
>>> t = (0,1,2,3,4,0,1,2)
>>> t.index(3)
3
>>> t.count(0)
2
```

(Contributed by Raymond Hettinger)

- The built-in types now have improved support for extended slicing syntax, accepting various combinations of `(start, stop, step)`. Previously, the support was partial and certain corner
cases wouldn’t work. (Implemented by Thomas Wouters.)

- Properties now have three attributes, getter, setter and deleter, that are decorators providing useful shortcuts for adding a getter, setter or deleter function to an existing property. You would use them like this:

```python
class C(object):
    @property
def x(self):
        return self._x

    @x.setter
def x(self, value):
        self._x = value

    @x.deleter
def x(self):
        del self._x

class D(C):
    @C.x.getter
def x(self):
        return self._x * 2

    @x.setter
def x(self, value):
        self._x = value / 2
```

- Several methods of the built-in set types now accept multiple iterables: intersection(), intersection_update(), union(), update(), difference() and difference_update().

```python
>>> s=set('1234567890')
```
Many floating-point features were added. The `float()` function will now turn the string `nan` into an IEEE 754 Not A Number value, and `+inf` and `-inf` into positive or negative infinity. This works on any platform with IEEE 754 semantics. (Contributed by Christian Heimes; bpo-1635.)

Other functions in the `math` module, `isinf()` and `isnan()`, return true if their floating-point argument is infinite or Not A Number. (bpo-1640)

Conversion functions were added to convert floating-point numbers into hexadecimal strings (bpo-3008). These functions convert floats to and from a string representation without introducing rounding errors from the conversion between decimal and binary. Floats have a `hex()` method that returns a string representation, and the `float.fromhex()` method converts a string back into a number:

```python
>>> a = 3.75
>>> a.hex()
'0x1.e000000000000p+1'
>>> float.fromhex('0x1.e000000000000p+1')
3.75
>>> b=1./3
>>> b.hex()
'0x1.55555555555555p-2'
```
A numerical nicety: when creating a complex number from two floats on systems that support signed zeros (-0 and +0), the \texttt{complex()} constructor will now preserve the sign of the zero. (Fixed by Mark T. Dickinson; \texttt{bpo-1507}.)

Classes that inherit a \texttt{__hash__()} method from a parent class can set \texttt{__hash__ = None} to indicate that the class isn't hashable. This will make \texttt{hash(obj)} raise a \texttt{TypeError} and the class will not be indicated as implementing the Hashable \texttt{ABC}.

You should do this when you've defined a \texttt{__cmp__()} or \texttt{__eq__()} method that compares objects by their value rather than by identity. All objects have a default hash method that uses \texttt{id(obj)} as the hash value. There's no tidy way to remove the \texttt{__hash__()} method inherited from a parent class, so assigning \texttt{None} was implemented as an override. At the C level, extensions can set \texttt{tp_hash} to \texttt{PyObject_HashNotImplemented()}. (Fixed by Nick Coghlan and Amaury Forgeot d'Arc; \texttt{bpo-2235}.)

The \texttt{GeneratorExit} exception now subclasses \texttt{BaseException} instead of \texttt{Exception}. This means that an exception handler that does \texttt{except Exception:} will not inadvertently catch \texttt{GeneratorExit}. (Contributed by Chad Austin; \texttt{bpo-1537}.)

Generator objects now have a \texttt{gi_code} attribute that refers to the original code object backing the generator. (Contributed by Collin Winter; \texttt{bpo-1473257}.)

The \texttt{compile()} built-in function now accepts keyword arguments as well as positional parameters. (Contributed by}
The `complex()` constructor now accepts strings containing parenthesized complex numbers, meaning that `complex(repr(cplx))` will now round-trip values. For example, `complex('(3+4j)')` now returns the value `(3+4j)`.

The string `translate()` method now accepts `None` as the translation table parameter, which is treated as the identity transformation. This makes it easier to carry out operations that only delete characters. (Contributed by Bengt Richter and implemented by Raymond Hettinger; bpo-1193128.)

The built-in `dir()` function now checks for a `__dir__()` method on the objects it receives. This method must return a list of strings containing the names of valid attributes for the object, and lets the object control the value that `dir()` produces. Objects that have `__getattr__()` or `__getattribute__()` methods can use this to advertise pseudo-attributes they will honor. (bpo-1591665)

Instance method objects have new attributes for the object and function comprising the method; the new synonym for `im_self` is `__self__`, and `im_func` is also available as `__func__`. The old names are still supported in Python 2.6, but are gone in 3.0.

An obscure change: when you use the `locals()` function inside a `class` statement, the resulting dictionary no longer returns free variables. (Free variables, in this case, are variables referenced in the class statement that aren’t attributes of the class.)

Optimizations
The `warnings` module has been rewritten in C. This makes it possible to invoke warnings from the parser, and may also make the interpreter's startup faster. (Contributed by Neal Norwitz and Brett Cannon; bpo-1631171.)

Type objects now have a cache of methods that can reduce the work required to find the correct method implementation for a particular class; once cached, the interpreter doesn't need to traverse base classes to figure out the right method to call. The cache is cleared if a base class or the class itself is modified, so the cache should remain correct even in the face of Python's dynamic nature. (Original optimization implemented by Armin Rigo, updated for Python 2.6 by Kevin Jacobs; bpo-1700288.)

By default, this change is only applied to types that are included with the Python core. Extension modules may not necessarily be compatible with this cache, so they must explicitly add `Py_TPFLAGS_HAVE_VERSION_TAG` to the module's `tp_flags` field to enable the method cache. (To be compatible with the method cache, the extension module's code must not directly access and modify the `tp_dict` member of any of the types it implements. Most modules don't do this, but it's impossible for the Python interpreter to determine that. See bpo-1878 for some discussion.)

Function calls that use keyword arguments are significantly faster by doing a quick pointer comparison, usually saving the time of a full string comparison. (Contributed by Raymond Hettinger, after an initial implementation by Antoine Pitrou; bpo-1819.)

All of the functions in the `struct` module have been rewritten in C, thanks to work at the Need For Speed sprint. (Contributed by Raymond Hettinger.)
Some of the standard built-in types now set a bit in their type objects. This speeds up checking whether an object is a subclass of one of these types. (Contributed by Neal Norwitz.)

Unicode strings now use faster code for detecting whitespace and line breaks; this speeds up the split() method by about 25% and splitlines() by 35%. (Contributed by Antoine Pitrou.) Memory usage is reduced by using pymalloc for the Unicode string's data.

The with statement now stores the __exit__() method on the stack, producing a small speedup. (Implemented by Jeffrey Yasskin.)

To reduce memory usage, the garbage collector will now clear internal free lists when garbage-collecting the highest generation of objects. This may return memory to the operating system sooner.

Interpreter Changes

Two command-line options have been reserved for use by other Python implementations. The -J switch has been reserved for use by Jython for Jython-specific options, such as switches that are passed to the underlying JVM. -X has been reserved for options specific to a particular implementation of Python such as CPython, Jython, or IronPython. If either option is used with Python 2.6, the interpreter will report that the option isn't currently used.

Python can now be prevented from writing .pyc or .pyo files by supplying the -B switch to the Python interpreter, or by setting the
**PYTHONDONTWRITEBYTECODE** environment variable before running the interpreter. This setting is available to Python programs as the `sys.dont_write_bytecode` variable, and Python code can change the value to modify the interpreter’s behaviour. (Contributed by Neal Norwitz and Georg Brandl.)

The encoding used for standard input, output, and standard error can be specified by setting the **PYTHONIOENCODING** environment variable before running the interpreter. The value should be a string in the form `<encoding>` or `<encoding>:=<errorhandler>. The *encoding* part specifies the encoding's name, e.g. utf-8 or latin-1; the optional *errorhandler* part specifies what to do with characters that can't be handled by the encoding, and should be one of “error”, “ignore”, or “replace”. (Contributed by Martin von Löwis.)
New and Improved Modules

As in every release, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The **asyncore** and **asynchat** modules are being actively maintained again, and a number of patches and bugfixes were applied. (Maintained by Josiah Carlson; see bpo-1736190 for one patch.)

- The **bsddb** module also has a new maintainer, Jesús Cea Avión, and the package is now available as a standalone package. The web page for the package is www.jcea.es/programacion/pybsddb.htm. The plan is to remove the package from the standard library in Python 3.0, because its pace of releases is much more frequent than Python’s.

  The **bsddb.dbshelve** module now uses the highest pickling protocol available, instead of restricting itself to protocol 1. (Contributed by W. Barnes.)

- The **cgi** module will now read variables from the query string of an HTTP POST request. This makes it possible to use form actions with URLs that include query strings such as “/cgi-bin/add.py?category=1”. (Contributed by Alexandre Fiori and Nubis; bpo-1817.)

  The **parse_qs()** and **parse_qsl()** functions have been relocated from the **cgi** module to the **urlparse** module. The
versions still available in the cgi module will trigger PendingDeprecationWarning messages in 2.6 (bpo-600362).

- The cmath module underwent extensive revision, contributed by Mark Dickinson and Christian Heimes. Five new functions were added:
  - `polar()` converts a complex number to polar form, returning the modulus and argument of the complex number.
  - `rect()` does the opposite, turning a modulus, argument pair back into the corresponding complex number.
  - `phase()` returns the argument (also called the angle) of a complex number.
  - `isnan()` returns True if either the real or imaginary part of its argument is a NaN.
  - `isinf()` returns True if either the real or imaginary part of its argument is infinite.

The revisions also improved the numerical soundness of the cmath module. For all functions, the real and imaginary parts of the results are accurate to within a few units of least precision (ulps) whenever possible. See bpo-1381 for the details. The branch cuts for asinh(), atanh(): and atan() have also been corrected.

The tests for the module have been greatly expanded; nearly 2000 new test cases exercise the algebraic functions.

On IEEE 754 platforms, the cmath module now handles IEEE 754 special values and floating-point exceptions in a manner consistent with Annex ‘G’ of the C99 standard.

- A new data type in the collections module:
namedtuple(typename, fieldnames) is a factory function that creates subclasses of the standard tuple whose fields are accessible by name as well as by index. For example:

```python
>>> var_type = collections.namedtuple('variable', ...
...   'id name type size')
>>> # Names are separated by spaces or commas.
>>> # 'id, name, type, size' would also work.
>>> var_type._fields
('id', 'name', 'type', 'size')

>>> var = var_type(1, 'frequency', 'int', 4)
>>> print var[0], var.id    # Equivalent
1 1
>>> print var[2], var.type  # Equivalent
int int
>>> var._asdict()
{'size': 4, 'type': 'int', 'id': 1, 'name': 'frequency'}
>>> v2 = var._replace(name='amplitude')
>>> v2
variable(id=1, name='amplitude', type='int', size=4)
```

Several places in the standard library that returned tuples have been modified to return namedtuple instances. For example, the Decimal.as_tuple() method now returns a named tuple with sign, digits, and exponent fields.

(Contributed by Raymond Hettinger.)

- Another change to the collections module is that the deque type now supports an optional maxlen parameter; if supplied, the deque’s size will be restricted to no more than maxlen items. Adding more items to a full deque causes old items to be discarded.
from collections import deque

dq = deque(maxlen=3)
dq = deque([], maxlen=3)
dq.append(1); dq.append(2); dq.append(3)
dq = deque([1, 2, 3], maxlen=3)
dq.append(4)
dq = deque([2, 3, 4], maxlen=3)

(Contributed by Raymond Hettinger.)

- The Cookie module's Morsel objects now support an `httponly` attribute. In some browsers, cookies with this attribute set cannot be accessed or manipulated by JavaScript code. (Contributed by Arvin Schnell; bpo-1638033.)

- A new window method in the `curses` module, `chgat()`, changes the display attributes for a certain number of characters on a single line. (Contributed by Fabian Kreutz.)

```python
# Boldface text starting at y=0, x=21
# and affecting the rest of the line.
stdscr.chgat(0, 21, curses.A_BOLD)
```

The Textbox class in the `curses.textpad` module now supports editing in insert mode as well as overwrite mode. Insert mode is enabled by supplying a true value for the `insert_mode` parameter when creating the Textbox instance.

- The `datetime` module's `strftime()` methods now support a `%f` format code that expands to the number of microseconds in the object, zero-padded on the left to six places. (Contributed by
The `decimal` module was updated to version 1.66 of the General Decimal Specification. New features include some methods for some basic mathematical functions such as `exp()` and `log10()`:

```python
g>>> Decimal(1).exp()  
Decimal("2.718281828459045235360287471")
>>> Decimal("2.7182818").ln()  
Decimal("0.999999895305022877376682436")
>>> Decimal(1000).log10()  
Decimal("3")
```

The `as_tuple()` method of Decimal objects now returns a named tuple with sign, digits, and exponent fields.

(Implemented by Facundo Batista and Mark Dickinson. Named tuple support added by Raymond Hettinger.)

- The `difflib` module's `SequenceMatcher` class now returns named tuples representing matches, with a, b, and size attributes. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the `ftplib.FTP` class constructor as well as the `connect()` method. (Added by Facundo Batista.) Also, the FTP class's `storbinary()` and `storlines()` now take an optional `callback` parameter that will be called with each block of data after the data has been sent. (Contributed by Phil Schwartz; `bpo-1221598`.)

- The `reduce()` built-in function is also available in the
**functools** module. In Python 3.0, the builtin has been dropped and `reduce()` is only available from **functools**; currently there are no plans to drop the builtin in the 2.x series. (Patched by Christian Heimes; bpo-1739906.)

- When possible, the **getpass** module will now use `/dev/tty` to print a prompt message and read the password, falling back to standard error and standard input. If the password may be echoed to the terminal, a warning is printed before the prompt is displayed. (Contributed by Gregory P. Smith.)

- The **glob.glob()** function can now return Unicode filenames if a Unicode path was used and Unicode filenames are matched within the directory. (bpo-1001604)

- A new function in the **heapq** module, `merge(iter1, iter2, ...)`, takes any number of iterables returning data in sorted order, and returns a new generator that returns the contents of all the iterators, also in sorted order. For example:

  ```python
  >>> list(heapq.merge([1, 3, 5, 9], [2, 8, 16]))
  [1, 2, 3, 5, 8, 9, 16]
  ```

  Another new function, `heappushpop(heap, item)`, pushes `item` onto `heap`, then pops off and returns the smallest item. This is more efficient than making a call to `heappush()` and then `heappop()`.

  **heapq** is now implemented to only use less-than comparison, instead of the less-than-or-equal comparison it previously used. This makes **heapq**'s usage of a type match the **list.sort()** method. (Contributed by Raymond Hettinger.)
• An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the `httplib.HTTPConnection` and `HTTPSConnection` class constructors. (Added by Facundo Batista.)

• Most of the `inspect` module's functions, such as `getmoduleinfo()` and `getargs()`, now return named tuples. In addition to behaving like tuples, the elements of the return value can also be accessed as attributes. (Contributed by Raymond Hettinger.)

Some new functions in the module include `isgenerator()`, `isgeneratorfunction()`, and `isabstract()`.

• The `itertools` module gained several new functions.

```
izip_longest(iter1, iter2, ...[, fillvalue])
```

makes tuples from each of the elements; if some of the iterables are shorter than others, the missing values are set to `fillvalue`. For example:

```python
>>> tuple(itertools.izip_longest([1,2,3], [1,2,3], ((1, 1), (2, 2), (3, 3), (None, 4), (None, 5)))
((1, 1), (2, 2), (3, 3), (None, 4), (None, 5))
```

```
product(iter1, iter2, ..., [repeat=N])
```

returns the Cartesian product of the supplied iterables, a set of tuples containing every possible combination of the elements returned from each iterable.

```python
>>> list(itertools.product([1,2,3], [4,5,6]))
[(1, 4), (1, 5), (1, 6),
 (2, 4), (2, 5), (2, 6),
 (3, 4), (3, 5), (3, 6)]
```
The optional *repeat* keyword argument is used for taking the product of an iterable or a set of iterables with themselves, repeated \( N \) times. With a single iterable argument, \( N \)-tuples are returned:

```python
>>> list(itertools.product([1,2], repeat=3))
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
```

With two iterables, \( 2N \)-tuples are returned.

```python
>>> list(itertools.product([1,2], [3,4], repeat=2))
[(1, 3, 1, 3), (1, 3, 1, 4), (1, 3, 2, 3), (1, 3, 2, 4), (1, 4, 1, 3), (1, 4, 1, 4), (1, 4, 2, 3), (1, 4, 2, 4), (2, 3, 1, 3), (2, 3, 1, 4), (2, 3, 2, 3), (2, 3, 2, 4), (2, 4, 1, 3), (2, 4, 1, 4), (2, 4, 2, 3), (2, 4, 2, 4)]
```

*combinations(iterable, \( r \))* returns sub-sequences of length \( r \) from the elements of *iterable*.

```python
>>> list(itertools.combinations('123', 2))
[('1', '2'), ('1', '3'), ('2', '3')]
>>> list(itertools.combinations('123', 3))
[('1', '2', '3')]
>>> list(itertools.combinations('1234', 3))
[('1', '2', '3'), ('1', '2', '4'), ('1', '3', '4'), ('2', '3', '4')]
```

*permutations(iter[, \( r \)])* returns all the permutations of length \( r \) of the iterable's elements. If \( r \) is not specified, it will default to the number of elements produced by the iterable.

```python
>>> list(itertools.permutations([1,2,3,4], 2))
```
itertools.chain(*iterables) is an existing function in itertools that gained a new constructor in Python 2.6. itertools.chain.from_iterable(iterable) takes a single iterable that should return other iterables. chain() will then return all the elements of the first iterable, then all the elements of the second, and so on.

```python
>>> list(itertools.chain.from_iterable([[1,2,3], [1, 2, 3, 4, 5, 6]])
```

(All contributed by Raymond Hettinger.)

- The logging module's FileHandler class and its subclasses WatchedFileHandler, RotatingFileHandler, and TimedRotatingFileHandler now have an optional delay parameter to their constructors. If delay is true, opening of the log file is deferred until the first emit() call is made. (Contributed by Vinay Sajip.)

TimedRotatingFileHandler also has a utc constructor parameter. If the argument is true, UTC time will be used in determining when midnight occurs and in generating filenames; otherwise local time will be used.

- Several new functions were added to the math module:
  - isnan() and isinfin() determine whether a given float is a 
    (positive or negative) infinity or a NaN (Not a Number),

```python
[(1, 2), (1, 3), (1, 4),
 (2, 1), (2, 3), (2, 4),
 (3, 1), (3, 2), (3, 4),
 (4, 1), (4, 2), (4, 3)]
```
respectively.

- `copysign()` copies the sign bit of an IEEE 754 number, returning the absolute value of \( x \) combined with the sign bit of \( y \). For example, `math.copysign(1, -0.0)` returns -1.0. (Contributed by Christian Heimes.)

- `factorial()` computes the factorial of a number. (Contributed by Raymond Hettinger; `bpo-2138`.)

- `fsum()` adds up the stream of numbers from an iterable, and is careful to avoid loss of precision through using partial sums. (Contributed by Jean Brouwers, Raymond Hettinger, and Mark Dickinson; `bpo-2819`.)

- `acosh()`, `asinh()` and `atanh()` compute the inverse hyperbolic functions.

- `log1p()` returns the natural logarithm of \( 1+x \) (base e).

- `trunc()` rounds a number toward zero, returning the closest `Integral` that's between the function's argument and zero. Added as part of the backport of PEP 3141's type hierarchy for numbers.

- The `math` module has been improved to give more consistent behaviour across platforms, especially with respect to handling of floating-point exceptions and IEEE 754 special values.

Whenever possible, the module follows the recommendations of the C99 standard about 754's special values. For example, `sqrt(-1.)` should now give a `ValueError` across almost all platforms, while `sqrt(float('NaN'))` should return a NaN on all IEEE 754 platforms. Where Annex ‘F’ of the C99 standard recommends signaling ‘divide-by-zero’ or ‘invalid’, Python will raise `ValueError`. Where Annex ‘F’ of the C99 standard recommends signaling ‘overflow’, Python will raise `OverflowError`. (See `bpo-711019` and `bpo-1640`.)

(Contributed by Christian Heimes and Mark Dickinson.)
• **mmap** objects now have a `rfind()` method that searches for a substring beginning at the end of the string and searching backwards. The `find()` method also gained an `end` parameter giving an index at which to stop searching. (Contributed by John Lenton.)

• The **operator** module gained a `methodcaller()` function that takes a name and an optional set of arguments, returning a callable that will call the named function on any arguments passed to it. For example:

```python
>>> # Equivalent to lambda s: s.replace('old', 'new')
>>> replacer = operator.methodcaller('replace', 'old')
>>> replacer('old wine in old bottles')
'new wine in new bottles'
```

(Contributed by Georg Brandl, after a suggestion by Gregory Petrosyan.)

The `attrgetter()` function now accepts dotted names and performs the corresponding attribute lookups:

```python
>>> inst_name = operator.attrgetter(...
...    '__class__.__name__')
>>> inst_name('')
'str'
>>> inst_name(help)
 '_Helper'
```

(Contributed by Georg Brandl, after a suggestion by Barry Warsaw.)

• The **os** module now wraps several new system calls.
fchmod(fd, mode) and fchown(fd, uid, gid) change the mode and ownership of an opened file, and lchmod(path, mode) changes the mode of a symlink. (Contributed by Georg Brandl and Christian Heimes.)

chflags() and lchflags() are wrappers for the corresponding system calls (where they're available), changing the flags set on a file. Constants for the flag values are defined in the stat module; some possible values include UF_IMMUTABLE to signal the file may not be changed and UF_APPEND to indicate that data can only be appended to the file. (Contributed by M. Levinson.)

os.closerange(low, high) efficiently closes all file descriptors from low to high, ignoring any errors and not including high itself. This function is now used by the subprocess module to make starting processes faster. (Contributed by Georg Brandl; bpo-1663329.)

- The os.environ object's clear() method will now unset the environment variables using os.unsetenv() in addition to clearing the object's keys. (Contributed by Martin Horcicka; bpo-1181.)

- The os.walk() function now has a followlinks parameter. If set to True, it will follow symlinks pointing to directories and visit the directory's contents. For backward compatibility, the parameter's default value is false. Note that the function can fall into an infinite recursion if there's a symlink that points to a parent directory. (bpo-1273829)

- In the os.path module, the splitext() function has been
changed to not split on leading period characters. This produces better results when operating on Unix's dot-files. For example, `os.path.splitext('.ipython')` now returns `('.ipython', '')` instead of `('', '.ipython')`. (bpo-1115886)

A new function, `os.path.relpath(path, start='.')`, returns a relative path from the `start` path, if it's supplied, or from the current working directory to the destination `path`. (Contributed by Richard Barran; bpo-1339796.)

On Windows, `os.path.expandvars()` will now expand environment variables given in the form “%var%”, and “~user” will be expanded into the user’s home directory path. (Contributed by Josiah Carlson; bpo-957650.)

- The Python debugger provided by the `pdb` module gained a new command: “run” restarts the Python program being debugged and can optionally take new command-line arguments for the program. (Contributed by Rocky Bernstein; bpo-1393667.)

- The `pdb.post_mortem()` function, used to begin debugging a traceback, will now use the traceback returned by `sys.exc_info()` if no traceback is supplied. (Contributed by Facundo Batista; bpo-1106316.)

- The `pickletools` module now has an `optimize()` function that takes a string containing a pickle and removes some unused opcodes, returning a shorter pickle that contains the same data structure. (Contributed by Raymond Hettinger.)

- A `get_data()` function was added to the `pkgutil` module that returns the contents of resource files included with an installed
Python package. For example:

```python
>>> import pkgutil
>>> print(pkgutil.get_data('test', 'exception_hierarchy.txt'))
BaseException
   +-- SystemExit
   +-- KeyboardInterrupt
   +-- GeneratorExit
   +-- Exception
      +-- StopIteration
      +-- StandardError
...
```

(Contributed by Paul Moore; bpo-2439.)

- The `pyexpat` module's Parser objects now allow setting their `buffer_size` attribute to change the size of the buffer used to hold character data. (Contributed by Achim Gaedke; bpo-1137.)

- The `Queue` module now provides queue variants that retrieve entries in different orders. The `PriorityQueue` class stores queued items in a heap and retrieves them in priority order, and `LifoQueue` retrieves the most recently added entries first, meaning that it behaves like a stack. (Contributed by Raymond Hettinger.)

- The `random` module's Random objects can now be pickled on a 32-bit system and unpickled on a 64-bit system, and vice versa. Unfortunately, this change also means that Python 2.6's Random objects can't be unpickled correctly on earlier versions of Python. (Contributed by Shawn Ligocki; bpo-1727780.)

The new `triangular(low, high, mode)` function returns
random numbers following a triangular distribution. The returned values are between low and high, not including high itself, and with mode as the most frequently occurring value in the distribution. (Contributed by Wladmir van der Laan and Raymond Hettinger; bpo-1681432.)

- Long regular expression searches carried out by the re module will check for signals being delivered, so time-consuming searches can now be interrupted. (Contributed by Josh Hoyt and Ralf Schmitt; bpo-846388.)

The regular expression module is implemented by compiling bytecodes for a tiny regex-specific virtual machine. Untrusted code could create malicious strings of bytecode directly and cause crashes, so Python 2.6 includes a verifier for the regex bytecode. (Contributed by Guido van Rossum from work for Google App Engine; bpo-3487.)

- The rlcompleter module's Completer.complete() method will now ignore exceptions triggered while evaluating a name. (Fixed by Lorenz Quack; bpo-2250.)

- The sched module's scheduler instances now have a read-only queue attribute that returns the contents of the scheduler's queue, represented as a list of named tuples with the fields (time, priority, action, argument). (Contributed by Raymond Hettinger; bpo-1861.)

- The select module now has wrapper functions for the Linux epoll() and BSD kqueue() system calls. modify() method was added to the existing poll objects; pollobj.modify(fd, eventmask) takes a file descriptor or file object and an event
mask, modifying the recorded event mask for that file. (Contributed by Christian Heimes; bpo-1657.)

- The `shutil.copytree()` function now has an optional `ignore` argument that takes a callable object. This callable will receive each directory path and a list of the directory's contents, and returns a list of names that will be ignored, not copied.

The `shutil` module also provides an `ignore_patterns()` function for use with this new parameter. `ignore_patterns()` takes an arbitrary number of glob-style patterns and returns a callable that will ignore any files and directories that match any of these patterns. The following example copies a directory tree, but skips both `.svn` directories and Emacs backup files, which have names ending with `~`:

```python
shutil.copytree('Doc/library', '/tmp/library', ignore=shutil.ignore_patterns('*.~'))
```

(Contributed by Tarek Ziadé; bpo-2663.)

- Integrating signal handling with GUI handling event loops like those used by Tkinter or GTK+ has long been a problem; most software ends up polling, waking up every fraction of a second to check if any GUI events have occurred. The `signal` module can now make this more efficient. Calling `signal.set_wakeup_fd(fd)` sets a file descriptor to be used; when a signal is received, a byte is written to that file descriptor. There's also a C-level function, `PySignal_SetWakeupFd()`, for setting the descriptor.

Event loops will use this by opening a pipe to create two
descriptors, one for reading and one for writing. The writable
descriptor will be passed to set_wakeup_fd(), and the
readable descriptor will be added to the list of descriptors
monitored by the event loop via select() or poll(). On
receiving a signal, a byte will be written and the main event loop
will be woken up, avoiding the need to poll.

(Contributed by Adam Olsen; bpo-1583.)

The siginterrupt() function is now available from Python
code, and allows changing whether signals can interrupt system
calls or not. (Contributed by Ralf Schmitt.)

The setitimer() and getitimer() functions have also been
added (where they're available). setitimer() allows setting
interval timers that will cause a signal to be delivered to the
process after a specified time, measured in wall-clock time,
consumed process time, or combined process+system time.
(Contributed by Guilherme Polo; bpo-2240.)

- The smtplib module now supports SMTP over SSL thanks to
the addition of the SMTP_SSL class. This class supports an
interface identical to the existing SMTP class. (Contributed by
Monty Taylor.) Both class constructors also have an optional
timeout parameter that specifies a timeout for the initial
connection attempt, measured in seconds. (Contributed by
Facundo Batista.)

An implementation of the LMTP protocol (RFC 2033) was also
added to the module. LMTP is used in place of SMTP when
transferring e-mail between agents that don't manage a mail
queue. (LMTP implemented by Leif Hedstrom; bpo-957003.)
SMTP.starttls() now complies with RFC 3207 and forgets any knowledge obtained from the server not obtained from the TLS negotiation itself. (Patch contributed by Bill Fenner; bpo-829951.)

- The socket module now supports TIPC (http://tipc.sourceforge.net/), a high-performance non-IP-based protocol designed for use in clustered environments. TIPC addresses are 4- or 5-tuples. (Contributed by Alberto Bertogli; bpo-1646.)

A new function, create_connection(), takes an address and connects to it using an optional timeout value, returning the connected socket object. This function also looks up the address’s type and connects to it using IPv4 or IPv6 as appropriate. Changing your code to use create_connection() instead of socket(socket.AF_INET, ...) may be all that’s required to make your code work with IPv6.

- The base classes in the SocketServer module now support calling a handle_timeout() method after a span of inactivity specified by the server’s timeout attribute. (Contributed by Michael Pomraning.) The serve_forever() method now takes an optional poll interval measured in seconds, controlling how often the server will check for a shutdown request. (Contributed by Pedro Werneck and Jeffrey Yasskin; bpo-742598, bpo-1193577.)

- The sqlite3 module, maintained by Gerhard Häring, has been updated from version 2.3.2 in Python 2.5 to version 2.4.1.

- The struct module now supports the C99 _Bool type, using
the format character ' ?'. (Contributed by David Remahl.)

- The Popen objects provided by the subprocess module now have terminate(), kill(), and send_signal() methods. On Windows, send_signal() only supports the SIGTERM signal, and all these methods are aliases for the Win32 API function TerminateProcess(). (Contributed by Christian Heimes.)

- A new variable in the sys module, float_info, is an object containing information derived from the float.h file about the platform's floating-point support. Attributes of this object include mant_dig (number of digits in the mantissa), epsilon (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; bpo-1534.)

Another new variable, dont_write_bytecode, controls whether Python writes any .pyc or .pyo files on importing a module. If this variable is true, the compiled files are not written. The variable is initially set on start-up by supplying the -B switch to the Python interpreter, or by setting the PYTHONDONOTWRITEBYTECODE environment variable before running the interpreter. Python code can subsequently change the value of this variable to control whether bytecode files are written or not. (Contributed by Neal Norwitz and Georg Brandl.)

Information about the command-line arguments supplied to the Python interpreter is available by reading attributes of a named tuple available as sys.flags. For example, the verbose attribute is true if Python was executed in verbose mode, debug is true in debugging mode, etc. These attributes are all read-only. (Contributed by Christian Heimes.)
A new function, `getsizeof()`, takes a Python object and returns the amount of memory used by the object, measured in bytes. Built-in objects return correct results; third-party extensions may not, but can define a `__sizeof__()` method to return the object's size. (Contributed by Robert Schuppenies; bpo-2898.)

It's now possible to determine the current profiler and tracer functions by calling `sys.getprofile()` and `sys.gettrace()`.(Contributed by Georg Brandl; bpo-1648.)

- The `tarfile` module now supports POSIX.1-2001 (pax) tarfiles in addition to the POSIX.1-1988 (ustar) and GNU tar formats that were already supported. The default format is GNU tar; specify the `format` parameter to open a file using a different format:

```python
import tarfile

with tarfile.open('output.tar', 'w', format=tarfile.PAX_FORMAT) as tar:
    tar.add('example.txt')
```

The new `encoding` and `errors` parameters specify an encoding and an error handling scheme for character conversions. `'strict'`, `'ignore'`, and `'replace'` are the three standard ways Python can handle errors; `'utf-8'` is a special value that replaces bad characters with their UTF-8 representation. (Character conversions occur because the PAX format supports Unicode filenames, defaulting to UTF-8 encoding.)

The `TarFile.add()` method now accepts an `exclude` argument that's a function that can be used to exclude certain filenames from an archive. The function must take a filename and return `true` if the file should be excluded or `false` if it should be archived. The function is applied to both the name initially passed
to add() and to the names of files in recursively-added directories.

(All changes contributed by Lars Gustäbel).

- An optional `timeout` parameter was added to the `telnetlib.Telnet` class constructor, specifying a timeout measured in seconds. (Added by Facundo Batista.)

- The `tempfile.NamedTemporaryFile` class usually deletes the temporary file it created when the file is closed. This behaviour can now be changed by passing `delete=False` to the constructor. (Contributed by Damien Miller; bpo-1537850.)

A new class, `SpooledTemporaryFile`, behaves like a temporary file but stores its data in memory until a maximum size is exceeded. On reaching that limit, the contents will be written to an on-disk temporary file. (Contributed by Dustin J. Mitchell.)

The `NamedTemporaryFile` and `SpooledTemporaryFile` classes both work as context managers, so you can write with

```python
with tempfile.NamedTemporaryFile() as tmp:
    ....
```

(Contributed by Alexander Belopolsky; bpo-2021.)

- The `test.test_support` module gained a number of context managers useful for writing tests. `EnvironmentVarGuard()` is a context manager that temporarily changes environment variables and automatically restores them to their old values.

Another context manager, `TransientResource`, can surround calls to resources that may or may not be available; it will catch and ignore a specified list of exceptions. For example, a network test may ignore certain failures when connecting to an external
Finally, check_warnings() resets the warning module's warning filters and returns an object that will record all warning messages triggered (bpo-3781):

```
with test_support.check_warnings() as wrec:
    warnings.simplefilter("always")
    # ... code that triggers a warning ...
    assert str(wrec.message) == "function is outdated"
    assert len(wrec.warnings) == 1, "Multiple warnings"
```

(Contributed by Brett Cannon.)

- The **textwrap** module can now preserve existing whitespace at the beginnings and ends of the newly-created lines by specifying `drop_whitespace=False` as an argument:

```
>>> S = """This sentence has a bunch of ... extra whitespace.""
>>> print textwrap.fill(S, width=15)
This sentence has a bunch of extra whitespace.
>>> print textwrap.fill(S, drop_whitespace=False)
This sentence has a bunch of extra whitespace.
```
The `threading` module API is being changed to use properties such as `daemon` instead of `setDaemon()` and `isDaemon()` methods, and some methods have been renamed to use underscores instead of camel-case; for example, the `activeCount()` method is renamed to `active_count()`. Both the 2.6 and 3.0 versions of the module support the same properties and renamed methods, but don't remove the old methods. No date has been set for the deprecation of the old APIs in Python 3.x; the old APIs won't be removed in any 2.x version. (Carried out by several people, most notably Benjamin Peterson.)

The `threading` module's Thread objects gained an `ident` property that returns the thread's identifier, a nonzero integer. (Contributed by Gregory P. Smith; bpo-2871.)

The `timeit` module now accepts callables as well as strings for the statement being timed and for the setup code. Two convenience functions were added for creating Timer instances: `repeat(stmt, setup, time, repeat, number)` and `timeit(stmt, setup, time, number)` create an instance and call the corresponding method. (Contributed by Erik Demaine; bpo-1533909.)

The Tkinter module now accepts lists and tuples for options, separating the elements by spaces before passing the resulting value to Tcl/Tk. (Contributed by Guilherme Polo; bpo-2906.)
The `turtle` module for turtle graphics was greatly enhanced by Gregor Lingl. New features in the module include:

- Better animation of turtle movement and rotation.
- Control over turtle movement using the new `delay()`, `tracer()`, and `speed()` methods.
- The ability to set new shapes for the turtle, and to define a new coordinate system.
- Turtles now have an `undo()` method that can roll back actions.
- Simple support for reacting to input events such as mouse and keyboard activity, making it possible to write simple games.
- A `turtle.cfg` file can be used to customize the starting appearance of the turtle's screen.
- The module's docstrings can be replaced by new docstrings that have been translated into another language.

(bpo-1513695)

- An optional `timeout` parameter was added to the `urllib.urlopen()` function and the `urllib.ftpwrapper` class constructor, as well as the `urllib2.urlopen()` function. The parameter specifies a timeout measured in seconds. For example:

```python
>>> u = urllib2.urlopen("http://slow.example.com", timeout=3)
Traceback (most recent call last):
  ... 
urllib2.URLError: <urlopen error timed out>
>>>
```
The Unicode database provided by the `unicodedata` module has been updated to version 5.1.0. (Updated by Martin von Löwis; `bpo-3811`.)

The `warnings` module's `formatwarning()` and `showwarning()` gained an optional `line` argument that can be used to supply the line of source code. (Added as part of `bpo-1631171`, which re-implemented part of the `warnings` module in C code.)

A new function, `catch_warnings()`, is a context manager intended for testing purposes that lets you temporarily modify the warning filters and then restore their original values (`bpo-3781`).

The XML-RPC `SimpleXMLRPCServer` and `DocXMLRPCServer` classes can now be prevented from immediately opening and binding to their socket by passing `False` as the `bind_and_activate` constructor parameter. This can be used to modify the instance's `allow_reuse_address` attribute before calling the `server_bind()` and `server_activate()` methods to open the socket and begin listening for connections. (Contributed by Peter Parente; `bpo-1599845`.)

`SimpleXMLRPCServer` also has a `_send_traceback_header` attribute; if true, the exception and formatted traceback are returned as HTTP headers “X-Exception” and “X-Traceback”. This feature is for debugging purposes only and should not be used on production servers because the tracebacks might reveal passwords or other sensitive information. (Contributed by Alan McIntyre as part of his project for Google's)
Summer of Code 2007.)

- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using `xmlrpclib` should convert `date` and `time` instances. ([bpo-1330538](https://bugs.python.org/issue1330538)) The code can also handle dates before 1900 (contributed by Ralf Schmitt; [bpo-2014](https://bugs.python.org/issue2014)) and 64-bit integers represented by using `<i8>` in XML-RPC responses (contributed by Riku Lindblad; [bpo-2985](https://bugs.python.org/issue2985)).

- The `zipfile` module's ZipFile class now has `extract()` and `extractall()` methods that will unpack a single file or all the files in the archive to the current directory, or to a specified directory:

```python
z = zipfile.ZipFile('python-251.zip')
# Unpack a single file, writing it relative
# to the /tmp directory.
z.extract('Python/sysmodule.c', '/tmp')

# Unpack all the files in the archive.
z.extractall()
```

(Contributed by Alan McIntyre; [bpo-467924](https://bugs.python.org/issue467924).)

The `open()`, `read()` and `extract()` methods can now take either a filename or a `ZipInfo` object. This is useful when an archive accidentally contains a duplicated filename. (Contributed by Graham Horler; [bpo-1775025](https://bugs.python.org/issue1775025).)
Finally, *zipfile* now supports using Unicode filenames for archived files. (Contributed by Alexey Borzenkov; [bpo-1734346](https://bugs.python.org/issue1734346).)

The **ast** module

The **ast** module provides an Abstract Syntax Tree representation of Python code, and Armin Ronacher contributed a set of helper functions that perform a variety of common tasks. These will be useful for HTML templating packages, code analyzers, and similar tools that process Python code.

The `parse()` function takes an expression and returns an AST. The `dump()` function outputs a representation of a tree, suitable for debugging:

```python
import ast
t = ast.parse(""
    d = {}
    for i in 'abcdefgijklm':
        d[i + i] = ord(i) - ord('a') + 1
print d
""")
print ast.dump(t)
```

This outputs a deeply nested tree:

```python
Module(body=[
    Assign(targets=[
        Name(id='d', ctx=Store())
    ], value=Dict(keys=[], values=[]))
    For(target=Name(id='i', ctx=Store()),
        iter=Str(s='abcdefgijklm'), body=[
            Assign(targets=]
```
The `literal_eval()` method takes a string or an AST representing a literal expression, parses and evaluates it, and returns the resulting value. A literal expression is a Python expression containing only strings, numbers, dictionaries, etc. but no statements or function calls. If you need to evaluate an expression but cannot accept the security risk of using an `eval()` call, `literal_eval()` will handle it safely:

```python
>>> literal = '(
    "a", 
    "b", 
    {2:4, 3:8, 1:2})'
>>> print ast.literal_eval(literal)
('a', 'b', {1: 2, 2: 4, 3: 8})
>>> print ast.literal_eval("a" + "b")
```
The module also includes NodeVisitor and NodeTransformer classes for traversing and modifying an AST, and functions for common transformations such as changing line numbers.

The **future_builtins** module

Python 3.0 makes many changes to the repertoire of built-in functions, and most of the changes can’t be introduced in the Python 2.x series because they would break compatibility. The **future_builtins** module provides versions of these built-in functions that can be imported when writing 3.0-compatible code.

The functions in this module currently include:

- **ascii(obj)**: equivalent to **repr()**. In Python 3.0, **repr()** will return a Unicode string, while **ascii()** will return a pure ASCII bytestring.
- **filter(predicate, iterable)**, **map(func, iterable1, ...)**: the 3.0 versions return iterators, unlike the 2.x builtins which return lists.
- **hex(value)**, **oct(value)**: instead of calling the **__hex__()** or **__oct__()** methods, these versions will call the **__index__()** method and convert the result to hexadecimal or octal. **oct()** will use the new 0o notation for its result.

The **json** module: JavaScript Object Notation

The new **json** module supports the encoding and decoding of Python...
types in JSON (Javascript Object Notation). JSON is a lightweight interchange format often used in web applications. For more information about JSON, see http://www.json.org.

json comes with support for decoding and encoding most built-in Python types. The following example encodes and decodes a dictionary:

```python
>>> import json
>>> data = {"spam": "foo", "parrot": 42}
>>> in_json = json.dumps(data) # Encode the data
>>> in_json
'{"parrot": 42, "spam": "foo"}'
>>> json.loads(in_json) # Decode into a Python object
{"spam": "foo", "parrot": 42}
```

It's also possible to write your own decoders and encoders to support more types. Pretty-printing of the JSON strings is also supported.

json (originally called simplejson) was written by Bob Ippolito.

The plistlib module: A Property-List Parser

The .plist format is commonly used on Mac OS X to store basic data types (numbers, strings, lists, and dictionaries) by serializing them into an XML-based format. It resembles the XML-RPC serialization of data types.

Despite being primarily used on Mac OS X, the format has nothing Mac-specific about it and the Python implementation works on any platform that Python supports, so the plistlib module has been promoted to the standard library.
Using the module is simple:

```python
import sys
import plistlib
import datetime

data_struct = dict(lastAccessed=datetime.datetime.now(),
                    version=1,
                    categories=('Personal', 'Shared', 'Private'))

plist_str = plistlib.writePlistToString(data_struct)
new_struct = plistlib.readPlistFromString(plist_str)
print(data_struct)
print(new_struct)

# Write data structure to a file and read it back.
plistlib.writePlist(data_struct, '/tmp/customizations.plist')
new_struct = plistlib.readPlist('/tmp/customizations.plist')

# read/writePlist accepts file-like objects as well
plistlib.writePlist(data_struct, sys.stdout)
```

**ctypes Enhancements**

Thomas Heller continued to maintain and enhance the `ctypes` module.

`ctypes` now supports a `c_bool` datatype that represents the C99 `bool` type. (Contributed by David Remahl; [bpo-1649190](https://bugs.python.org/issue1649190).)

The `ctypes` string, buffer and array types have improved support for extended slicing syntax, where various combinations of `(start,`
stop, step) are supplied. (Implemented by Thomas Wouters.)

All ctypes data types now support from_buffer() and from_buffer_copy() methods that create a ctypes instance based on a provided buffer object. from_buffer_copy() copies the contents of the object, while from_buffer() will share the same memory area.

A new calling convention tells ctypes to clear the errno or Win32 LastError variables at the outset of each wrapped call. (Implemented by Thomas Heller; bpo-1798.)

You can now retrieve the Unix errno variable after a function call. When creating a wrapped function, you can supply use_errno=True as a keyword parameter to the DLL() function and then call the module-level methods set_errno() and get_errno() to set and retrieve the error value.

The Win32 LastError variable is similarly supported by the DLL(), OleDLL(), and WinDLL() functions. You supply use_last_error=True as a keyword parameter and then call the module-level methods set_last_error() and get_last_error().

The byref() function, used to retrieve a pointer to a ctypes instance, now has an optional offset parameter that is a byte count that will be added to the returned pointer.

**Improved SSL Support**

Bill Janssen made extensive improvements to Python 2.6’s support for the Secure Sockets Layer by adding a new module, ssl, that's built
atop the OpenSSL library. This new module provides more control over the protocol negotiated, the X.509 certificates used, and has better support for writing SSL servers (as opposed to clients) in Python. The existing SSL support in the socket module hasn’t been removed and continues to work, though it will be removed in Python 3.0.

To use the new module, you must first create a TCP connection in the usual way and then pass it to the ssl.wrap_socket() function. It’s possible to specify whether a certificate is required, and to obtain certificate info by calling the getpeercert() method.

See also: The documentation for the ssl module.
Deprecations and Removals

- String exceptions have been removed. Attempting to use them raises a `TypeError`.

- Changes to the `Exception` interface as dictated by PEP 352 continue to be made. For 2.6, the message attribute is being deprecated in favor of the `args` attribute.

- (3.0-warning mode) Python 3.0 will feature a reorganized standard library that will drop many outdated modules and rename others. Python 2.6 running in 3.0-warning mode will warn about these modules when they are imported.

  The list of deprecated modules is: audiodev, bgenlocations, buildtools, bundlebuilder, Canvas, compiler, dircache, dl, fpformat, gensuitemodule, ihooks, imageop, imgfile, linuxaudiodev, mhlib, mimetools, multifile, new, pure, statvfs, sunaudiodev, test.testall, and toaiff.

- The gopherlib module has been removed.

- The MimeWriter module and mimify module have been deprecated; use the `email` package instead.

- The md5 module has been deprecated; use the `hashlib` module instead.

- The posixfile module has been deprecated; `fcntl.lockf()` provides better locking.
- The popen2 module has been deprecated; use the subprocess module.

- The rgbimg module has been removed.

- The sets module has been deprecated; it’s better to use the built-in set and frozenset types.

- The sha module has been deprecated; use the hashlib module instead.
Changes to Python’s build process and to the C API include:

- **Python now must be compiled with C89 compilers (after 19 years!).** This means that the Python source tree has dropped its own implementations of `memmove()` and `strerror()`, which are in the C89 standard library.

- **Python 2.6 can be built with Microsoft Visual Studio 2008 (version 9.0), and this is the new default compiler.** See the `PCbuild` directory for the build files. (Implemented by Christian Heimes.)

- **On Mac OS X, Python 2.6 can be compiled as a 4-way universal build.** The `configure` script can take a `--with-universal-archs=[32-bit|64-bit|all]` switch, controlling whether the binaries are built for 32-bit architectures (x86, PowerPC), 64-bit (x86-64 and PPC-64), or both. (Contributed by Ronald Oussoren.)

- The BerkeleyDB module now has a C API object, available as `bsddb.db.api`. This object can be used by other C extensions that wish to use the bsddb module for their own purposes. (Contributed by Duncan Grisby.)

- The new buffer interface, previously described in the PEP 3118 section, adds `PyObject_GetBuffer()` and `PyBuffer_Release()`, as well as a few other functions.

- **Python’s use of the C stdio library is now thread-safe, or at least as thread-safe as the underlying library is.** A long-standing potential bug occurred if one thread closed a file object while another thread was reading from or writing to the object. In 2.6 file
objects have a reference count, manipulated by the `PyFile_IncUseCount()` and `PyFile_DecUseCount()` functions. File objects can’t be closed unless the reference count is zero. `PyFile_IncUseCount()` should be called while the GIL is still held, before carrying out an I/O operation using the `FILE *` pointer, and `PyFile_DecUseCount()` should be called immediately after the GIL is re-acquired. (Contributed by Antoine Pitrou and Gregory P. Smith.)

- Importing modules simultaneously in two different threads no longer deadlocks; it will now raise an `ImportError`. A new API function, `PyImport_ImportModuleNoBlock()`, will look for a module in `sys.modules` first, then try to import it after acquiring an import lock. If the import lock is held by another thread, an `ImportError` is raised. (Contributed by Christian Heimes.)

- Several functions return information about the platform’s floating-point support. `PyFloat_GetMax()` returns the maximum representable floating point value, and `PyFloat_GetMin()` returns the minimum positive value. `PyFloat_GetInfo()` returns an object containing more information from the `float.h` file, such as "mant_dig" (number of digits in the mantissa), "epsilon" (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; bpo-1534.)

- C functions and methods that use `PyComplex_AsCComplex()` will now accept arguments that have a `__complex__()` method. In particular, the functions in the `cmath` module will now accept objects with this method. This is a backport of a Python 3.0 change. (Contributed by Mark Dickinson; bpo-1675423.)
Python’s C API now includes two functions for case-insensitive string comparisons, `PyOS_stricmp(char*, char*)` and `PyOS_strnicmp(char*, char*, Py_ssize_t)`. (Contributed by Christian Heimes; bpo-1635.)

Many C extensions define their own little macro for adding integers and strings to the module’s dictionary in the `init*` function. Python 2.6 finally defines standard macros for adding values to a module, `PyModule_AddStringMacro` and `PyModule_AddIntMacro()`. (Contributed by Christian Heimes.)

Some macros were renamed in both 3.0 and 2.6 to make it clearer that they are macros, not functions. `Py_Size()` became `Py_SIZE()`, `Py_Type()` became `Py_TYPE()`, and `Py_Refcnt()` became `Py_REFCNT()`. The mixed-case macros are still available in Python 2.6 for backward compatibility. (bpo-1629)

Distutils now places C extensions it builds in a different directory when running on a debug version of Python. (Contributed by Collin Winter; bpo-1530959.)

Several basic data types, such as integers and strings, maintain internal free lists of objects that can be re-used. The data structures for these free lists now follow a naming convention: the variable is always named `free_list`, the counter is always named `numfree`, and a macro `Py<typename>_MAXFREELIST` is always defined.

A new Makefile target, “make patchcheck”, prepares the Python source tree for making a patch: it fixes trailing whitespace in all modified `.py` files, checks whether the documentation has been
changed, and reports whether the Misc/ACKS and Misc/NEWS files have been updated. (Contributed by Brett Cannon.)

Another new target, “make profile-opt”, compiles a Python binary using GCC’s profile-guided optimization. It compiles Python with profiling enabled, runs the test suite to obtain a set of profiling results, and then compiles using these results for optimization. (Contributed by Gregory P. Smith.)

Port-Specific Changes: Windows

- The support for Windows 95, 98, ME and NT4 has been dropped. Python 2.6 requires at least Windows 2000 SP4.

- The new default compiler on Windows is Visual Studio 2008 (version 9.0). The build directories for Visual Studio 2003 (version 7.1) and 2005 (version 8.0) were moved into the PC/ directory. The new PCbuild directory supports cross compilation for X64, debug builds and Profile Guided Optimization (PGO). PGO builds are roughly 10% faster than normal builds. (Contributed by Christian Heimes with help from Amaury Forgeot d’Arc and Martin von Löwis.)

- The msvcrt module now supports both the normal and wide char variants of the console I/O API. The getwch() function reads a keypress and returns a Unicode value, as does the getwche() function. The putwch() function takes a Unicode character and writes it to the console. (Contributed by Christian Heimes.)

- os.path.expandvars() will now expand environment variables in the form “%var%”, and “~user” will be expanded into the user’s home directory path. (Contributed by Josiah Carlson;
The socket module's socket objects now have an ioctl() method that provides a limited interface to the WSAIoctl() system interface.

The _winreg module now has a function, ExpandEnvironmentStrings(), that expands environment variable references such as %NAME% in an input string. The handle objects provided by this module now support the context protocol, so they can be used in with statements. (Contributed by Christian Heimes.)

_winreg also has better support for x64 systems, exposing the DisableReflectionKey(), EnableReflectionKey(), and QueryReflectionKey() functions, which enable and disable registry reflection for 32-bit processes running on 64-bit systems. (bpo-1753245)

The msilib module's Record object gained GetInteger() and GetString() methods that return field values as an integer or a string. (Contributed by Floris Bruynooghe; bpo-2125.)

Port-Specific Changes: Mac OS X

When compiling a framework build of Python, you can now specify the framework name to be used by providing the --with-framework-name= option to the configure script.

The macfs module has been removed. This in turn required the macostools.touched() function to be removed because it depended on the macfs module. (bpo-1490190)

Many other Mac OS modules have been deprecated and will be
removed in Python 3.0: _builtinSuites, aepack, aetools, aetypes, applesingle, appletrawmain, appletrunner, argvemulator, Audio_mac, autoGIL, Carbon, cfmfile, CodeWarrior, ColorPicker, EasyDialogs, Explorer, Finder, FrameWork, findertools, ic, icglue, icopen, macerrors, MacOS, macfs, macostools, macresource, MiniAEFrame, Nav, Netscape, OSA Terminology, pimp, PixMapWrapper, StdSuites, SystemEvents, Terminal, and terminalcommand.

**Port-Specific Changes: IRIX**

A number of old IRIX-specific modules were deprecated and will be removed in Python 3.0: al and AL, cd, cddb, cdplayer, CL and cl, DEVICE, ERRNO, FILE, FL and fl, flp, fm, GET, GLWS, GL and gl, IN, IOCTL, jpeg, panelparser, readcd, SV and sv, torgb, videoreader, and WAIT.
Porting to Python 2.6

This section lists previously described changes and other bugfixes that may require changes to your code:

- Classes that aren’t supposed to be hashable should set \_\_hash\_\_ = None in their definitions to indicate the fact.

- String exceptions have been removed. Attempting to use them raises a TypeError.

- The \_\_init\_\_() method of collections.deque now clears any existing contents of the deque before adding elements from the iterable. This change makes the behavior match list.\_\_init\_\_().

- object.\_\_init\_\_() previously accepted arbitrary arguments and keyword arguments, ignoring them. In Python 2.6, this is no longer allowed and will result in a TypeError. This will affect \_\_init\_\_() methods that end up calling the corresponding method on object (perhaps through using super()). See bpo-1683368 for discussion.

- The Decimal constructor now accepts leading and trailing whitespace when passed a string. Previously it would raise an InvalidOperation exception. On the other hand, the create_decimal() method of Context objects now explicitly disallows extra whitespace, raising a ConversionSyntax exception.

- Due to an implementation accident, if you passed a file path to the built-in \_\_import\_\_() function, it would actually import the
specified file. This was never intended to work, however, and the implementation now explicitly checks for this case and raises an `ImportError`.

- C API: the `PyImport_Import()` and `PyImport_ImportModule()` functions now default to absolute imports, not relative imports. This will affect C extensions that import other modules.

- C API: extension data types that shouldn't be hashable should define their `tp_hash` slot to `PyObject_HashNotImplemented()`.

- The `socket` module exception `socket.error` now inherits from `IOError`. Previously it wasn't a subclass of `StandardError` but now it is, through `IOError`. (Implemented by Gregory P. Smith; bpo-1706815.)

- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using `xmlrpclib` should convert date and time instances. (bpo-1330538)

- (3.0-warning mode) The `Exception` class now warns when accessed using slicing or index access; having `Exception` behave like a tuple is being phased out.

- (3.0-warning mode) Inequality comparisons between two dictionaries or two objects that don't implement comparison methods are reported as warnings. `dict1 == dict2` still works, but `dict1 < dict2` is being phased out.
Comparisons between cells, which are an implementation detail of Python’s scoping rules, also cause warnings because such comparisons are forbidden entirely in 3.0.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Steve Brown, Nick Coghlan, Ralph Corderoy, Jim Jewett, Kent Johnson, Chris Lambacher, Martin Michlmayr, Antoine Pitrou, Brian Warner.
What’s New in Python 2.5

Author: A.M. Kuchling

This article explains the new features in Python 2.5. The final release of Python 2.5 is scheduled for August 2006; PEP 356 describes the planned release schedule.

The changes in Python 2.5 are an interesting mix of language and library improvements. The library enhancements will be more important to Python's user community, I think, because several widely-useful packages were added. New modules include ElementTree for XML processing (xml.etree), the SQLite database module (sqlite), and the ctypes module for calling C functions.

The language changes are of middling significance. Some pleasant new features were added, but most of them aren’t features that you'll use every day. Conditional expressions were finally added to the language using a novel syntax; see section PEP 308: Conditional Expressions. The new ‘with’ statement will make writing cleanup code easier (section PEP 343: The ‘with’ statement). Values can now be passed into generators (section PEP 342: New Generator Features). Imports are now visible as either absolute or relative (section PEP 328: Absolute and Relative Imports). Some corner cases of exception handling are handled better (section PEP 341: Unified try/except/finally). All these improvements are worthwhile, but they’re improvements to one specific language feature or another; none of them are broad modifications to Python’s semantics.

As well as the language and library additions, other improvements and bugfixes were made throughout the source tree. A search through the
SVN change logs finds there were 353 patches applied and 458 bugs fixed between Python 2.4 and 2.5. (Both figures are likely to be underestimates.)

This article doesn’t try to be a complete specification of the new features; instead changes are briefly introduced using helpful examples. For full details, you should always refer to the documentation for Python 2.5 at https://docs.python.org. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

Comments, suggestions, and error reports for this document are welcome; please e-mail them to the author or open a bug in the Python bug tracker.
PEP 308: Conditional Expressions

For a long time, people have been requesting a way to write conditional expressions, which are expressions that return value A or value B depending on whether a Boolean value is true or false. A conditional expression lets you write a single assignment statement that has the same effect as the following:

```python
if condition:
    x = true_value
else:
    x = false_value
```

There have been endless tedious discussions of syntax on both python-dev and comp.lang.python. A vote was even held that found the majority of voters wanted conditional expressions in some form, but there was no syntax that was preferred by a clear majority. Candidates included C's `cond ? true_v : false_v`, `if cond then true_v else false_v`, and 16 other variations.

Guido van Rossum eventually chose a surprising syntax:

```python
x = true_value if condition else false_value
```

Evaluation is still lazy as in existing Boolean expressions, so the order of evaluation jumps around a bit. The `condition` expression in the middle is evaluated first, and the `true_value` expression is evaluated only if the condition was true. Similarly, the `false_value` expression is only evaluated when the condition is false.

This syntax may seem strange and backwards; why does the condition go in the *middle* of the expression, and not in the front as in C's `x ?`
The decision was checked by applying the new syntax to the modules in the standard library and seeing how the resulting code read. In many cases where a conditional expression is used, one value seems to be the ‘common case’ and one value is an ‘exceptional case’, used only on rarer occasions when the condition isn’t met. The conditional syntax makes this pattern a bit more obvious:

```
color = ((doc + '
') if doc else '')
```

I read the above statement as meaning “here contents is usually assigned a value of doc+'\n'; sometimes doc is empty, in which special case an empty string is returned.” I doubt I will use conditional expressions very often where there isn’t a clear common and uncommon case.

There was some discussion of whether the language should require surrounding conditional expressions with parentheses. The decision was made to not require parentheses in the Python language’s grammar, but as a matter of style I think you should always use them. Consider these two statements:

```
# First version -- no parens
level = 1 if logging else 0

# Second version -- with parens
level = (1 if logging else 0)
```

In the first version, I think a reader’s eye might group the statement into ‘level = 1’, ‘if logging’, ‘else 0’, and think that the condition decides whether the assignment to level is performed. The second version reads better, in my opinion, because it makes it clear that the assignment is always performed and the choice is being made between two values.
Another reason for including the brackets: a few odd combinations of list comprehensions and lambdas could look like incorrect conditional expressions. See PEP 308 for some examples. If you put parentheses around your conditional expressions, you won’t run into this case.

See also:

PEP 308 - Conditional Expressions

PEP written by Guido van Rossum and Raymond D. Hettinger; implemented by Thomas Wouters.
PEP 309: Partial Function Application

The `functools` module is intended to contain tools for functional-style programming.

One useful tool in this module is the `partial()` function. For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function \( f(a, b, c) \); you could create a new function \( g(b, c) \) that was equivalent to \( f(1, \; b, \; c) \). This is called “partial function application”.

`partial()` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```python
import functools

def log(message, subsystem):
    "Write the contents of 'message' to the specified subsystem.
    print '%s: %s' % (subsystem, message)
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

Here's another example, from a program that uses PyGTK. Here a context-sensitive pop-up menu is being constructed dynamically. The callback provided for the menu option is a partially applied version of
the open_item() method, where the first argument has been provided.

```python
...  
class Application:
    def open_item(self, path):
        ...
    def init (self):
        open_func = functools.partial(self.open_item)
        popup_menu.append( ("Open", open_func, 1) )
```

Another function in the `functools` module is the `update_wrapper(wrapper, wrapped)` function that helps you write well-behaved decorators. `update_wrapper()` copies the name, module, and docstring attribute to a wrapper function so that tracebacks inside the wrapped function are easier to understand. For example, you might write:

```python
def my_decorator(f):
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    functools.update_wrapper(wrapper, f)
    return wrapper
```

`wraps()` is a decorator that can be used inside your own decorators to copy the wrapped function’s information. An alternate version of the previous example would be:

```python
def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)```
return wrapper

See also:

**PEP 309 - Partial Function Application**

PEP proposed and written by Peter Harris; implemented by Hye-Shik Chang and Nick Coghlan, with adaptations by Raymond Hettinger.
PEP 314: Metadata for Python Software Packages v1.1

Some simple dependency support was added to Distutils. The `setup()` function now has `requires`, `provides`, and `obsoletes` keyword parameters. When you build a source distribution using the `sdist` command, the dependency information will be recorded in the `PKG-INFO` file.

Another new keyword parameter is `download_url`, which should be set to a URL for the package's source code. This means it's now possible to look up an entry in the package index, determine the dependencies for a package, and download the required packages.

```
VERSION = '1.0'
ssetup(name='PyPackage',
    version=VERSION,
    requires=[['numarray', 'zlib (>=1.1.4)'],
    obsoletes=[OldPackage]
    download_url=('http://www.example.com/pypackage' % VERSION),
    )
```

Another new enhancement to the Python package index at [https://pypi.org](https://pypi.org) is storing source and binary archives for a package. The new `upload` Distutils command will upload a package to the repository.

Before a package can be uploaded, you must be able to build a distribution using the `sdist` Distutils command. Once that works, you can run `python setup.py upload` to add your package to the
PyPI archive. Optionally you can GPG-sign the package by supplying the --sign and --identity options.

Package uploading was implemented by Martin von Löwis and Richard Jones.

**See also:**

**PEP 314 - Metadata for Python Software Packages v1.1**
PEP proposed and written by A.M. Kuchling, Richard Jones, and Fred Drake; implemented by Richard Jones and Fred Drake.
PEP 328: Absolute and Relative Imports

The simpler part of PEP 328 was implemented in Python 2.4: parentheses could now be used to enclose the names imported from a module using the `from ... import ...` statement, making it easier to import many different names.

The more complicated part has been implemented in Python 2.5: importing a module can be specified to use absolute or package-relative imports. The plan is to move toward making absolute imports the default in future versions of Python.

Let's say you have a package directory like this:

```
pkg/
pkg/__init__.py
pkg/main.py
pkg/string.py
```

This defines a package named `pkg` containing the `pkg.main` and `pkg.string` submodules.

Consider the code in the `main.py` module. What happens if it executes the statement `import string`? In Python 2.4 and earlier, it will first look in the package's directory to perform a relative import, finds `pkg/string.py`, imports the contents of that file as the `pkg.string` module, and that module is bound to the name `string` in the `pkg.main` module's namespace.

That's fine if `pkg.string` was what you wanted. But what if you wanted Python's standard `string` module? There's no clean way to ignore `pkg.string` and look for the standard module; generally you
had to look at the contents of `sys.modules`, which is slightly unclean. Holger Krekel's `py.std` package provides a tidier way to perform imports from the standard library, `import py; py.std.string.join()`, but that package isn't available on all Python installations.

Reading code which relies on relative imports is also less clear, because a reader may be confused about which module, `string` or `pkg.string`, is intended to be used. Python users soon learned not to duplicate the names of standard library modules in the names of their packages' submodules, but you can't protect against having your submodule's name being used for a new module added in a future version of Python.

In Python 2.5, you can switch `import`'s behaviour to absolute imports using a `from __future__ import absolute_import` directive. This absolute-import behaviour will become the default in a future version (probably Python 2.7). Once absolute imports are the default, `import string` will always find the standard library's version. It's suggested that users should begin using absolute imports as much as possible, so it's preferable to begin writing `from pkg import string` in your code.

Relative imports are still possible by adding a leading period to the module name when using the `from ... import` form:

```python
# Import names from pkg.string
from .string import name1, name2
# Import pkg.string
from . import string
```

This imports the `string` module relative to the current package, so in
pkg.main this will import name1 and name2 from pkg.string. Additional leading periods perform the relative import starting from the parent of the current package. For example, code in the A.B.C module can do:

```python
from . import D          # Imports A.B.D
from .. import E        # Imports A.E
from ... import G       # Imports A.F.G
```

Leading periods cannot be used with the `import modname` form of the import statement, only the `from ... import` form.

See also:

**PEP 328 - Imports: Multi-Line and Absolute/Relative**

PEP written by Aahz; implemented by Thomas Wouters.

[https://pylib.readthedocs.io/](https://pylib.readthedocs.io/)

The py library by Holger Krekel, which contains the py.std package.
PEP 338: Executing Modules as Scripts

The `-m` switch added in Python 2.4 to execute a module as a script gained a few more abilities. Instead of being implemented in C code inside the Python interpreter, the switch now uses an implementation in a new module, `runpy`.

The `runpy` module implements a more sophisticated import mechanism so that it's now possible to run modules in a package such as `pychecker.checker`. The module also supports alternative import mechanisms such as the `zipimport` module. This means you can add a `.zip` archive's path to `sys.path` and then use the `-m` switch to execute code from the archive.

See also:

**PEP 338 - Executing modules as scripts**

PEP written and implemented by Nick Coghlan.
PEP 341: Unified try/except/finally

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn’t combine both `except` blocks and a `finally` block, because generating the right bytecode for the combined version was complicated and it wasn’t clear what the semantics of the combined statement should be.

Guido van Rossum spent some time working with Java, which does support the equivalent of combining `except` blocks and a `finally` block, and this clarified what the statement should mean. In Python 2.5, you can now write:

```python
try:
    block-1 ...
except Exception1:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code in `block-1` is executed. If the code raises an exception, the various `except` blocks are tested: if the exception is of class `Exception1`, `handler-1` is executed; otherwise if it’s of class `Exception2`, `handler-2` is executed, and so forth. If no exception is raised, the `else-block` is executed.
No matter what happened previously, the final-block is executed once the code block is complete and any raised exceptions handled. Even if there’s an error in an exception handler or the else-block and a new exception is raised, the code in the final-block is still run.

See also:

PEP 341 - Unifying try-except and try-finally
PEP written by Georg Brandl; implementation by Thomas Lee.
PEP 342: New Generator Features

Python 2.5 adds a simple way to pass values into a generator. As introduced in Python 2.3, generators only produce output; once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. Sometimes the ability to pass in some information would be useful. Hackish solutions to this include making the generator’s code look at a global variable and then changing the global variable’s value, or passing in some mutable object that callers then modify.

To refresh your memory of basic generators, here’s a simple example:

```python
def counter (maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1
```

When you call `counter(10)`, the result is an iterator that returns the values from 0 up to 9. On encountering the `yield` statement, the iterator returns the provided value and suspends the function's execution, preserving the local variables. Execution resumes on the following call to the iterator's `next()` method, picking up after the `yield` statement.

In Python 2.3, `yield` was a statement; it didn’t return any value. In 2.5, `yield` is now an expression, returning a value that can be assigned to a variable or otherwise operated on:

```python
val = (yield i)
```
I recommend that you always put parentheses around a `yield` expression when you’re doing something with the returned value, as in the above example. The parentheses aren’t always necessary, but it’s easier to always add them instead of having to remember when they’re needed.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there’s an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. The generator’s code is then resumed and the `yield` expression returns the specified `value`. If the regular `next()` method is called, the `yield` returns `None`.

Here’s the previous example, modified to allow changing the value of the internal counter.

```python
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here’s an example of changing the counter:

```python
>>> it = counter(10)
>>> print it.next()
```
yield will usually return None, so you should always check for this case. Don't just use its value in expressions unless you're sure that the send() method will be the only method used to resume your generator function.

In addition to send(), there are two other new methods on generators:

- throw(type, value=None, traceback=None) is used to raise an exception inside the generator; the exception is raised by the yield expression where the generator's execution is paused.

- close() raises a new GeneratorExit exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise GeneratorExit or StopIteration. Catching the GeneratorExit exception and returning a value is illegal and will trigger a RuntimeError; if the function raises some other exception, that exception is propagated to the caller. close() will also be called by Python's garbage collector when the generator is garbage-collected.
If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become *coroutines*, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements). We’ll have to figure out patterns for using coroutines effectively in Python.

The addition of the `close()` method has one side effect that isn’t obvious. `close()` is called when a generator is garbage-collected, so this means the generator’s code gets one last chance to run before the generator is destroyed. This last chance means that `try...finally` statements in generators can now be guaranteed to work; the `finally` clause will now always get a chance to run. The syntactic restriction that you couldn’t mix `yield` statements with a `try...finally` suite has therefore been removed. This seems like a minor bit of language trivia, but using generators and `try...finally` is actually necessary in order to implement the `with` statement described by PEP 343. I'll look at this new statement in the following section.

Another even more esoteric effect of this change: previously, the `gi_frame` attribute of a generator was always a frame object. It’s now possible for `gi_frame` to be `None` once the generator has been exhausted.
See also:

**PEP 342 - Coroutines via Enhanced Generators**

PEP written by Guido van Rossum and Phillip J. Eby; implemented by Phillip J. Eby. Includes examples of some fancier uses of generators as coroutines.

Earlier versions of these features were proposed in PEP 288 by Raymond Hettinger and PEP 325 by Samuele Pedroni.


The Wikipedia entry for coroutines.

http://www.sidhe.org/~dan/blog/archives/000178.html

An explanation of coroutines from a Perl point of view, written by Dan Sugalski.
PEP 343: The ‘with’ statement

The ‘with’ statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I'll discuss the statement as it will commonly be used. In the next section, I'll examine the implementation details and show how to write objects for use with this statement.

The ‘with’ statement is a new control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods.

The object's `__enter__()` is called before `with-block` is executed and therefore can run set-up code. It also may return a value that is bound to the name `variable`, if given. (Note carefully that `variable` is not assigned the result of `expression`.)

After execution of the `with-block` is finished, the object's `__exit__()` method is called, even if the block raised an exception, and can therefore run clean-up code.

To enable the statement in Python 2.5, you need to add the following directive to your module:

```
from __future__ import with_statement
```
The statement will always be enabled in Python 2.6.

Some standard Python objects now support the context management protocol and can be used with the `with` statement. File objects are one example:

```python
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
... more processing code ...
```

After this statement has executed, the file object in `f` will have been automatically closed, even if the `for` loop raised an exception part-way through the block.

**Note:** In this case, `f` is the same object created by `open()`, because `file.__enter__()` returns `self`.

The `threading` module’s locks and condition variables also support the `with` statement:

```python
lock = threading.Lock()
with lock:
    # Critical section of code
...
```

The lock is acquired before the block is executed and always released once the block is complete.

The new `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print v.sqrt()

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16
    # The original context is restored on exiting the block.
    print v.sqrt()

Writing Context Managers

Under the hood, the ‘with’ statement is fairly complicated. Most people will only use ‘with’ in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have \texttt{__enter__()} and \texttt{__exit__()} methods.
- The context manager’s \texttt{__enter__()} method is called. The value returned is assigned to \texttt{VAR}. If no \texttt{as VAR} clause is present, the value is simply discarded.
- The code in \texttt{BLOCK} is executed.
- If \texttt{BLOCK} raises an exception, the \texttt{__exit__(type, value, traceback)} is called with the exception details, the same values returned by \texttt{sys.exc_info()}. The method’s return value controls whether the exception is re-raised: any false value re-raises the exception, and \texttt{True} will result in suppressing it. You'll
only rarely want to suppress the exception, because if you do the
author of the code containing the ‘with’ statement will never
realize anything went wrong.
• If BLOCK didn’t raise an exception, the __exit__() method is
still called, but type, value, and traceback are all None.

Let’s think through an example. I won’t present detailed code but will
only sketch the methods necessary for a database that supports
transactions.

(For people unfamiliar with database terminology: a set of changes to
the database are grouped into a transaction. Transactions can be
either committed, meaning that all the changes are written into the
database, or rolled back, meaning that the changes are all discarded
and the database is unchanged. See any database textbook for more
information.)

Let’s assume there’s an object representing a database connection.
Our goal will be to let the user write code like this:

```
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute("insert into ...")
    cursor.execute("delete from ...")
# ... more operations ...
```

The transaction should be committed if the code in the block runs
flawlessly or rolled back if there’s an exception. Here’s the basic
interface for DatabaseConnection that I’ll assume:

```
class DatabaseConnection:
    # Database interface
def cursor (self):
    "Returns a cursor object and starts a new tr
```python
def commit (self):
    "Commits current transaction"
def rollback (self):
    "Rolls back current transaction"
```

The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add `as cursor` to their `with` statement to bind the cursor to a variable name.

```python
class DatabaseConnection:
    ...
def __enter__ (self):
    # Code to start a new transaction
cursor = self.cursor()
    return cursor
```

The `__exit__()` method is the most complicated because it’s where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of None. None is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.

```python
class DatabaseConnection:
    ...
def __exit__ (self, type, value, tb):
    if tb is None:
        # No exception, so commit
        self.commit()
```
else:
    # Exception occurred, so rollback.
    self.rollback()
    # return False

The contextlib module

The new contextlib module provides some functions and a decorator that are useful for writing objects for use with the `with` statement.

The decorator is called contextmanager(), and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the `yield` will be executed as the `__enter__()` method, and the value yielded will be the method's return value that will get bound to the variable in the `with` statement's as clause, if any. The code after the `yield` will be executed in the `__exit__()` method. Any exception raised in the block will be raised by the `yield` statement.

Our database example from the previous section could be written using this decorator as:

```python
from contextlib import contextmanager

@contextmanager
def db_transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
```
else:
    connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...

The `contextlib` module also has a `nested(mgr1, mgr2, ...)` function that combines a number of context managers so you don't need to write nested `with` statements. In this example, the single `with` statement both starts a database transaction and acquires a thread lock:

```python
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, lock):
    ...
```

Finally, the `closing(object)` function returns `object` so that it can be bound to a variable, and calls `object.close` at the end of the block.

```python
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)
```

See also:

**PEP 343 - The “with” statement**

PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal
Norwitz. The PEP shows the code generated for a ‘with’ statement, which can be helpful in learning how the statement works.

The documentation for the `contextlib` module.
PEP 352: Exceptions as New-Style Classes

Exception classes can now be new-style classes, not just classic classes, and the built-in Exception class and all the standard built-in exceptions (NameError, ValueError, etc.) are now new-style classes.

The inheritance hierarchy for exceptions has been rearranged a bit. In 2.5, the inheritance relationships are:

```
BaseException    # New in Python 2.5
 | - KeyboardInterrupt
 | - SystemExit
 | - Exception
    | - (all other current built-in exceptions)
```

This rearrangement was done because people often want to catch all exceptions that indicate program errors. KeyboardInterrupt and SystemExit aren’t errors, though, and usually represent an explicit action such as the user hitting Control-C or code calling sys.exit(). A bare except: will catch all exceptions, so you commonly need to list KeyboardInterrupt and SystemExit in order to re-raise them. The usual pattern is:

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    raise
except:
    # Log error...
    # Continue running program...
```
In Python 2.5, you can now write `except Exception` to achieve the same result, catching all the exceptions that usually indicate errors but leaving `KeyboardInterrupt` and `SystemExit` alone. As in previous versions, a bare `except:` still catches all exceptions.

The goal for Python 3.0 is to require any class raised as an exception to derive from `BaseException` or some descendant of `BaseException`, and future releases in the Python 2.x series may begin to enforce this constraint. Therefore, I suggest you begin making all your exception classes derive from `Exception` now. It’s been suggested that the bare `except:` form should be removed in Python 3.0, but Guido van Rossum hasn’t decided whether to do this or not.

Raising of strings as exceptions, as in the statement `raise "Error occurred"`, is deprecated in Python 2.5 and will trigger a warning. The aim is to be able to remove the string-exception feature in a few releases.

See also:

**PEP 352 - Required Superclass for Exceptions**  
PEP written by Brett Cannon and Guido van Rossum; implemented by Brett Cannon.
PEP 353: Using ssize_t as the index type

A wide-ranging change to Python’s C API, using a new Py_ssize_t type definition instead of int, will permit the interpreter to handle more data on 64-bit platforms. This change doesn’t affect Python’s capacity on 32-bit platforms.

Various pieces of the Python interpreter used C’s int type to store sizes or counts; for example, the number of items in a list or tuple were stored in an int. The C compilers for most 64-bit platforms still define int as a 32-bit type, so that meant that lists could only hold up to $2^{31} - 1 = 2147483647$ items. (There are actually a few different programming models that 64-bit C compilers can use – see http://www.unix.org/version2/whatsnew/lp64_wp.html for a discussion – but the most commonly available model leaves int as 32 bits.)

A limit of 2147483647 items doesn’t really matter on a 32-bit platform because you’ll run out of memory before hitting the length limit. Each list item requires space for a pointer, which is 4 bytes, plus space for a PyObject representing the item. 2147483647*4 is already more bytes than a 32-bit address space can contain.

It’s possible to address that much memory on a 64-bit platform, however. The pointers for a list that size would only require 16 GiB of space, so it’s not unreasonable that Python programmers might construct lists that large. Therefore, the Python interpreter had to be changed to use some type other than int, and this will be a 64-bit type on 64-bit platforms. The change will cause incompatibilities on 64-bit machines, so it was deemed worth making the transition now, while the number of 64-bit users is still relatively small. (In 5 or 10 years, we may all be on 64-bit machines, and the transition would be
more painful then.)

This change most strongly affects authors of C extension modules. Python strings and container types such as lists and tuples now use Py_ssize_t to store their size. Functions such as `PyList_Size()` now return Py_ssize_t. Code in extension modules may therefore need to have some variables changed to Py_ssize_t.

The `PyArg_ParseTuple()` and `Py_BuildValue()` functions have a new conversion code, n, for Py_ssize_t. `PyArg_ParseTuple()`'s s# and t# still output int by default, but you can define the macro PY_SSIZE_T_CLEAN before including Python.h to make them return Py_ssize_t.

**PEP 353** has a section on conversion guidelines that extension authors should read to learn about supporting 64-bit platforms.

See also:

**PEP 353** - Using ssize_t as the index type

PEP written and implemented by Martin von Löwis.
PEP 357: The ‘__index__’ method

The NumPy developers had a problem that could only be solved by adding a new special method, \texttt{__index__()}. When using slice notation, as in \texttt{[start:stop:step]}, the values of the \texttt{start}, \texttt{stop}, and \texttt{step} indexes must all be either integers or long integers. NumPy defines a variety of specialized integer types corresponding to unsigned and signed integers of 8, 16, 32, and 64 bits, but there was no way to signal that these types could be used as slice indexes.

Slicing can’t just use the existing \texttt{__int__()} method because that method is also used to implement coercion to integers. If slicing used \texttt{__int__()}, floating-point numbers would also become legal slice indexes and that’s clearly an undesirable behaviour.

Instead, a new special method called \texttt{__index__()} was added. It takes no arguments and returns an integer giving the slice index to use. For example:

```python
class C:
    def __index__(self):
        return self.value
```

The return value must be either a Python integer or long integer. The interpreter will check that the type returned is correct, and raises a \texttt{TypeError} if this requirement isn’t met.

A corresponding \texttt{nb\_index} slot was added to the C-level \texttt{PyNumberMethods} structure to let C extensions implement this protocol. \texttt{PyNumber\_Index(obj)} can be used in extension code to call the \texttt{__index__()} function and retrieve its result.
See also:

**PEP 357 - Allowing Any Object to be Used for Slicing**
PEP written and implemented by Travis Oliphant.
Other Language Changes

Here are all of the changes that Python 2.5 makes to the core Python language.

- The `dict` type has a new hook for letting subclasses provide a default value when a key isn't contained in the dictionary. When a key isn't found, the dictionary's `__missing__(key)` method will be called. This hook is used to implement the new `defaultdict` class in the `collections` module. The following example defines a dictionary that returns zero for any missing key:

```python
class zerodict (dict):
    def __missing__ (self, key):
        return 0

d = zerodict({1:1, 2:2})
print d[1], d[2]  # Prints 1, 2
print d[3], d[4]  # Prints 0, 0
```

- Both 8-bit and Unicode strings have new `partition(sep)` and `rpartition(sep)` methods that simplify a common use case.

  The `find(S)` method is often used to get an index which is then used to slice the string and obtain the pieces that are before and after the separator. `partition(sep)` condenses this pattern into a single method call that returns a 3-tuple containing the substring before the separator, the separator itself, and the substring after the separator. If the separator isn’t found, the first element of the tuple is the entire string and the other two elements are empty. `rpartition(sep)` also returns a 3-tuple but starts searching from the end of the string; the `r` stands for ‘reverse’.
Some examples:

```
>>> ('http://www.python.org').partition('://')
('http', '://', 'www.python.org')
>>> ('file:/usr/share/doc/index.html').partition('://')
('file:/usr/share/doc/index.html', '', '')
>>> (u'Subject: a quick question').partition(':')
(u'Subject', u':', u'a quick question')
>>> 'www.python.org'.rpartition('.')
('www.python', '.', 'org')
>>> 'www.python.org'.rpartition(':')
('', '', 'www.python.org')
```

(Implemented by Fredrik Lundh following a suggestion by Raymond Hettinger.)

- The `startswith()` and `endswith()` methods of string types now accept tuples of strings to check for.

```python
def is_image_file (filename):
    return filename.endswith(('gif', 'jpg', '.
```

(Implemented by Georg Brandl following a suggestion by Tom Lynn.)

- The `min()` and `max()` built-in functions gained a key keyword parameter analogous to the key argument for `sort()`. This parameter supplies a function that takes a single argument and is called for every value in the list; `min()`/`max()` will return the element with the smallest/largest return value from this function. For example, to find the longest string in a list, you can do:

```
L = ['medium', 'longest', 'short']
```
Two new built-in functions, `any()` and `all()`, evaluate whether an iterator contains any true or false values. `any()` returns `True` if any value returned by the iterator is true; otherwise it will return `False`. `all()` returns `True` only if all of the values returned by the iterator evaluate as true. (Suggested by Guido van Rossum, and implemented by Raymond Hettinger.)

The result of a class's `__hash__()` method can now be either a long integer or a regular integer. If a long integer is returned, the hash of that value is taken. In earlier versions the hash value was required to be a regular integer, but in 2.5 the `id()` built-in was changed to always return non-negative numbers, and users often seem to use `id(self)` in `__hash__()` methods (though this is discouraged).

ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error. See PEP 263 for how to declare a module's encoding; for example, you might add a line like this near the top of the source file:

```
# -*- coding: latin1 -*-
```

A new warning, `UnicodeWarning`, is triggered when you attempt
to compare a Unicode string and an 8-bit string that can’t be converted to Unicode using the default ASCII encoding. The result of the comparison is false:

```python
>>> chr(128) == unichr(128)  # Can't convert chr to Unicode
__main__:1: UnicodeWarning: Unicode equal comparison failed to convert both arguments to Unicode - interpreting them as being unequal
False
>>> chr(127) == unichr(127)  # chr(127) can be converted
True
```

Previously this would raise a `UnicodeDecodeError` exception, but in 2.5 this could result in puzzling problems when accessing a dictionary. If you looked up `unichr(128)` and `chr(128)` was being used as a key, you’d get a `UnicodeDecodeError` exception. Other changes in 2.5 resulted in this exception being raised instead of suppressed by the code in `dictobject.c` that implements dictionaries.

Raising an exception for such a comparison is strictly correct, but the change might have broken code, so instead `UnicodeWarning` was introduced.

(Implemented by Marc-André Lemburg.)

- One error that Python programmers sometimes make is forgetting to include an `__init__.py` module in a package directory. Debugging this mistake can be confusing, and usually requires running Python with the `-v` switch to log all the paths searched. In Python 2.5, a new `ImportWarning` warning is triggered when an import would have picked up a directory as a package but no
__init__.py was found. This warning is silently ignored by default; provide the -Wd option when running the Python executable to display the warning message. (Implemented by Thomas Wouters.)

- The list of base classes in a class definition can now be empty. As an example, this is now legal:

```python
class C():
    pass
```

(Implemented by Brett Cannon.)

**Interactive Interpreter Changes**

In the interactive interpreter, `quit` and `exit` have long been strings so that new users get a somewhat helpful message when they try to `quit`:

```python
>>> quit
'Use Ctrl-D (i.e. EOF) to exit.'
```

In Python 2.5, `quit` and `exit` are now objects that still produce string representations of themselves, but are also callable. Newbies who try `quit()` or `exit()` will now exit the interpreter as they expect. (Implemented by Georg Brandl.)

The Python executable now accepts the standard long options `--help` and `--version`; on Windows, it also accepts the `/?` option for displaying a help message. (Implemented by Georg Brandl.)

**Optimizations**
Several of the optimizations were developed at the NeedForSpeed sprint, an event held in Reykjavik, Iceland, from May 21–28 2006. The sprint focused on speed enhancements to the CPython implementation and was funded by EWT LLC with local support from CCP Games. Those optimizations added at this sprint are specially marked in the following list.

- When they were introduced in Python 2.4, the built-in `set` and `frozenset` types were built on top of Python’s dictionary type. In 2.5 the internal data structure has been customized for implementing sets, and as a result sets will use a third less memory and are somewhat faster. (Implemented by Raymond Hettinger.)
- The speed of some Unicode operations, such as finding substrings, string splitting, and character map encoding and decoding, has been improved. (Substring search and splitting improvements were added by Fredrik Lundh and Andrew Dalke at the NeedForSpeed sprint. Character maps were improved by Walter Dörwald and Martin von Löwis.)
- The `long(str, base)` function is now faster on long digit strings because fewer intermediate results are calculated. The peak is for strings of around 800–1000 digits where the function is 6 times faster. (Contributed by Alan McIntyre and committed at the NeedForSpeed sprint.)
- It’s now illegal to mix iterating over a file with `for line in file` and calling the file object’s `read()`, `readline()`, `readlines()` methods. Iteration uses an internal buffer and the `read*()` methods don’t use that buffer. Instead they would return the data following the buffer, causing the data to appear out of order. Mixing iteration and these methods will now trigger a `ValueError` from the `read*()` method. (Implemented by Thomas Wouters.)
- The `struct` module now compiles structure format strings into
an internal representation and caches this representation, yielding a 20% speedup. (Contributed by Bob Ippolito at the NeedForSpeed sprint.)

- The `re` module got a 1 or 2% speedup by switching to Python's allocator functions instead of the system's `malloc()` and `free()`. (Contributed by Jack Diederich at the NeedForSpeed sprint.)

- The code generator's peephole optimizer now performs simple constant folding in expressions. If you write something like `a = 2+3`, the code generator will do the arithmetic and produce code corresponding to `a = 5`. (Proposed and implemented by Raymond Hettinger.)

- Function calls are now faster because code objects now keep the most recently finished frame (a “zombie frame”) in an internal field of the code object, reusing it the next time the code object is invoked. (Original patch by Michael Hudson, modified by Armin Rigo and Richard Jones; committed at the NeedForSpeed sprint.)

- Frame objects are also slightly smaller, which may improve cache locality and reduce memory usage a bit. (Contributed by Neal Norwitz.)

- Python’s built-in exceptions are now new-style classes, a change that speeds up instantiation considerably. Exception handling in Python 2.5 is therefore about 30% faster than in 2.4. (Contributed by Richard Jones, Georg Brandl and Sean Reifschneider at the NeedForSpeed sprint.)

- Importing now caches the paths tried, recording whether they exist or not so that the interpreter makes fewer `open()` and `stat()` calls on startup. (Contributed by Martin von Löwis and Georg Brandl.)
New, Improved, and Removed Modules

The standard library received many enhancements and bug fixes in Python 2.5. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the SVN logs for all the details.

- The audioop module now supports the a-LAW encoding, and the code for u-LAW encoding has been improved. (Contributed by Lars Immisch.)

- The codecs module gained support for incremental codecs. The codec.lookup() function now returns a CodecInfo instance instead of a tuple. CodecInfo instances behave like a 4-tuple to preserve backward compatibility but also have the attributes encode, decode, incrementalencoder, incrementaldecoder, streamwriter, and streamreader. Incremental codecs can receive input and produce output in multiple chunks; the output is the same as if the entire input was fed to the non-incremental codec. See the codecs module documentation for details. (Designed and implemented by Walter Dörwald.)

- The collections module gained a new type, defaultdict, that subclasses the standard dict type. The new type mostly behaves like a dictionary but constructs a default value when a key isn't present, automatically adding it to the dictionary for the requested key value.

  The first argument to defaultdict's constructor is a factory
function that gets called whenever a key is requested but not found. This factory function receives no arguments, so you can use built-in type constructors such as `list()` or `int()`. For example, you can make an index of words based on their initial letter like this:

```python
words = """Nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura che la diritta via era smarrita""".lower().split()
index = defaultdict(list)
for w in words:
    init_letter = w[0]
    index[init_letter].append(w)
```

Printing `index` results in the following output:

```
defaultdict(<type 'list'>, {'c': ['cammin', 'che', 'd': ['del', 'di', 'diritta'], 'm': ['mezzo'], 'l': ['la'], 'o': ['oscura'], 'n': ['nel'], 'p': ['per'], 's': ['selva', 'smarrita'], 'r': ['ritrovai'], 'u': ['una'], 'v': ['vita']})
```

(Contributed by Guido van Rossum.)

- The `deque` double-ended queue type supplied by the `collections` module now has a `remove(value)` method that removes the first occurrence of `value` in the queue, raising `ValueError` if the value isn't found. (Contributed by Raymond Hettinger.)

- New module: The `contextlib` module contains helper functions
for use with the new 'with' statement. See section The contextlib module for more about this module.

- New module: The cProfile module is a C implementation of the existing profile module that has much lower overhead. The module's interface is the same as profile: you run cProfile.run('main()') to profile a function, can save profile data to a file, etc. It's not yet known if the Hotshot profiler, which is also written in C but doesn't match the profile module's interface, will continue to be maintained in future versions of Python. (Contributed by Armin Rigo.)

Also, the pstats module for analyzing the data measured by the profiler now supports directing the output to any file object by supplying a stream argument to the Stats constructor. (Contributed by Skip Montanaro.)

- The csv module, which parses files in comma-separated value format, received several enhancements and a number of bugfixes. You can now set the maximum size in bytes of a field by calling the csv.field_size_limit(new_limit) function; omitting the new_limit argument will return the currently-set limit. The reader class now has a line_num attribute that counts the number of physical lines read from the source; records can span multiple physical lines, so line_num is not the same as the number of records read.

The CSV parser is now stricter about multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files that contained carriage return characters within fields, so the
code was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner that preserves the newline characters.

(Contributed by Skip Montanaro and Andrew McNamara.)

- The `datetime` class in the `datetime` module now has a `strptime(string, format)` method for parsing date strings, contributed by Josh Spoerri. It uses the same format characters as `time.strptime()` and `time.strftime()`:

```python
from datetime import datetime

ts = datetime.strptime('10:13:15 2006-03-07', '%H:%M:%S %Y-%m-%d')
```

- The `SequenceMatcher.get_matching_blocks()` method in the `difflib` module now guarantees to return a minimal list of blocks describing matching subsequences. Previously, the algorithm would occasionally break a block of matching elements into two list entries. (Enhancement by Tim Peters.)

- The `doctest` module gained a `SKIP` option that keeps an example from being executed at all. This is intended for code snippets that are usage examples intended for the reader and aren’t actually test cases.

An `encoding` parameter was added to the `testfile()` function and the `DocFileSuite` class to specify the file's encoding. This makes it easier to use non-ASCII characters in tests contained within a docstring. (Contributed by Bjorn Tillenius.)
The email package has been updated to version 4.0. (Contributed by Barry Warsaw.)

The fileinput module was made more flexible. Unicode filenames are now supported, and a mode parameter that defaults to "r" was added to the input() function to allow opening files in binary or universal newlines mode. Another new parameter, openhook, lets you use a function other than open() to open the input files. Once you're iterating over the set of files, the FileInput object's new fileno() returns the file descriptor for the currently opened file. (Contributed by Georg Brandl.)

In the gc module, the new get_count() function returns a 3-tuple containing the current collection counts for the three GC generations. This is accounting information for the garbage collector; when these counts reach a specified threshold, a garbage collection sweep will be made. The existing gc.collect() function now takes an optional generation argument of 0, 1, or 2 to specify which generation to collect. (Contributed by Barry Warsaw.)

The nsmallest() and nlargest() functions in the heapq module now support a key keyword parameter similar to the one provided by the min()/max() functions and the sort() methods. For example:

```python
>>> import heapq
>>> L = ['short', 'medium', 'longest', 'longer still']
>>> heapq.nsmallest(2, L)  # Return two lowest elements
['longer still', 'longest']
>>> heapq.nsmallest(2, L, key=len)  # Return two shortest elements
['short', 'medium']
```
The `itertools.islice()` function now accepts None for the start and step arguments. This makes it more compatible with the attributes of slice objects, so that you can now write the following:

```python
s = slice(5)    # Create slice object
itertools.islice(iterable, s.start, s.stop, s.step)
```

The `format()` function in the `locale` module has been modified and two new functions were added, `format_string()` and `currency()`.

The `format()` function's `val` parameter could previously be a string as long as no more than one `%char` specifier appeared; now the parameter must be exactly one `%char` specifier with no surrounding text. An optional `monetary` parameter was also added which, if True, will use the locale's rules for formatting currency in placing a separator between groups of three digits.

To format strings with multiple `%char` specifiers, use the new `format_string()` function that works like `format()` but also supports mixing `%char` specifiers with arbitrary text.

A new `currency()` function was also added that formats a number according to the current locale's settings.

The `mailbox` module underwent a massive rewrite to add the capability to modify mailboxes in addition to reading them. A new
set of classes that include \texttt{mbox}, \texttt{MH}, and \texttt{Maildir} are used to read mailboxes, and have an \texttt{add} method to add messages, \texttt{remove} to remove messages, and \texttt{lock}/\texttt{unlock} to lock/unlock the mailbox. The following example converts a \texttt{maildir}-format mailbox into an \texttt{mbox}-format one:

```python
import mailbox

# 'factory=None' uses email.Message.Message as the class representing individual messages.
src = mailbox.Maildir('maildir', factory=None)
dest = mailbox.mbox('/tmp/mbox')

for msg in src:
    dest.add(msg)
```

(Contributed by Gregory K. Johnson. Funding was provided by Google's 2005 Summer of Code.)

- New module: the \texttt{msilib} module allows creating Microsoft Installer \texttt{.msi} files and CAB files. Some support for reading the \texttt{.msi} database is also included. (Contributed by Martin von Löwis.)

- The \texttt{nis} module now supports accessing domains other than the system default domain by supplying a \texttt{domain} argument to the \texttt{nis.match()} and \texttt{nis.maps()} functions. (Contributed by Ben Bell.)

- The \texttt{operator} module's \texttt{itemgetter()} and \texttt{attrgetter()} functions now support multiple fields. A call such as \texttt{operator.attrgetter('a', 'b')} will return a function that
retrieves the a and b attributes. Combining this new feature with the sort() method's key parameter lets you easily sort lists using multiple fields. (Contributed by Raymond Hettinger.)

- The optparse module was updated to version 1.5.1 of the Optik library. The OptionParser class gained an epilog attribute, a string that will be printed after the help message, and a destroy() method to break reference cycles created by the object. (Contributed by Greg Ward.)

- The os module underwent several changes. The stat_float_times variable now defaults to true, meaning that os.stat() will now return time values as floats. (This doesn't necessarily mean that os.stat() will return times that are precise to fractions of a second; not all systems support such precision.)

  Constants named os.SEEK_SET, os.SEEK_CUR, and os.SEEK_END have been added; these are the parameters to the os.lseek() function. Two new constants for locking are os.O_SHLOCK and os.O_EXLOCK.

  Two new functions, wait3() and wait4(), were added. They're similar the waitpid() function which waits for a child process to exit and returns a tuple of the process ID and its exit status, but wait3() and wait4() return additional information. wait3() doesn't take a process ID as input, so it waits for any child process to exit and returns a 3-tuple of process-id, exit-status, resource-usage as returned from the resource.getrusage() function. wait4(pid) does take a process ID. (Contributed by Chad J. Schroeder.)
On FreeBSD, the `os.stat()` function now returns times with nanosecond resolution, and the returned object now has `st_gen` and `st_birthtime`. The `st_flags` attribute is also available, if the platform supports it. (Contributed by Antti Louko and Diego Pettenò.)

- The Python debugger provided by the `pdb` module can now store lists of commands to execute when a breakpoint is reached and execution stops. Once breakpoint #1 has been created, enter `commands 1` and enter a series of commands to be executed, finishing the list with `end`. The command list can include commands that resume execution, such as `continue` or `next`. (Contributed by Grégoire Dooms.)

- The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The ability to return `None` was deprecated in Python 2.4, so this completes the removal of the feature.

- The `pkgutil` module, containing various utility functions for finding packages, was enhanced to support PEP 302’s import hooks and now also works for packages stored in ZIP-format archives. (Contributed by Phillip J. Eby.)

- The `pybench` benchmark suite by Marc-André Lemburg is now included in the `Tools/pybench` directory. The `pybench` suite is an improvement on the commonly used `pystone.py` program because `pybench` provides a more detailed measurement of the interpreter’s speed. It times particular operations such as function calls, tuple slicing, method lookups, and numeric operations, instead of performing many different operations and reducing the
result to a single number as pystone.py does.

- The pyexpat module now uses version 2.0 of the Expat parser. (Contributed by Trent Mick.)

- The Queue class provided by the Queue module gained two new methods. `join()` blocks until all items in the queue have been retrieved and all processing work on the items have been completed. Worker threads call the other new method, `task_done()`, to signal that processing for an item has been completed. (Contributed by Raymond Hettinger.)

- The old regex and regsub modules, which have been deprecated ever since Python 2.0, have finally been deleted. Other deleted modules: statcache, tzparse, whrandom.

- Also deleted: the `lib-old` directory, which includes ancient modules such as dircmp and ni, was removed. `lib-old` wasn't on the default `sys.path`, so unless your programs explicitly added the directory to `sys.path`, this removal shouldn't affect your code.

- The `rlcompleter` module is no longer dependent on importing the `readline` module and therefore now works on non-Unix platforms. (Patch from Robert Kiendl.)

- The SimpleXMLRPCServer and DocXMLRPCServer classes now have a `rpc_paths` attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/` and `'/RPC2'`. Setting `rpc_paths` to `None` or an empty tuple disables this path checking.
The **socket** module now supports AF_NETLINK sockets on Linux, thanks to a patch from Philippe Biondi. Netlink sockets are a Linux-specific mechanism for communications between a user-space process and kernel code; an introductory article about them is at [https://www.linuxjournal.com/article/7356](https://www.linuxjournal.com/article/7356). In Python code, netlink addresses are represented as a tuple of 2 integers, \((\text{pid}, \text{group	extunderscore mask})\).

Two new methods on socket objects, `recv_into(buffer)` and `recvfrom_into(buffer)`, store the received data in an object that supports the buffer protocol instead of returning the data as a string. This means you can put the data directly into an array or a memory-mapped file.

Socket objects also gained `getfamily()`, `gettype()`, and `getproto()` accessor methods to retrieve the family, type, and protocol values for the socket.

New module: the **spwd** module provides functions for accessing the shadow password database on systems that support shadow passwords.

The **struct** is now faster because it compiles format strings into `Struct` objects with `pack()` and `unpack()` methods. This is similar to how the `re` module lets you create compiled regular expression objects. You can still use the module-level `pack()` and `unpack()` functions; they'll create `Struct` objects and cache them. Or you can use `Struct` instances directly:

```python
s = struct.Struct('ih3s')
data = s.pack(1972, 187, 'abc')
```
year, number, name = s.unpack(data)

You can also pack and unpack data to and from buffer objects directly using the `pack_into(buffer, offset, v1, v2, ...)` and `unpack_from(buffer, offset)` methods. This lets you store data directly into an array or a memory-mapped file.

(Struct objects were implemented by Bob Ippolito at the NeedForSpeed sprint. Support for buffer objects was added by Martin Blais, also at the NeedForSpeed sprint.)

- The Python developers switched from CVS to Subversion during the 2.5 development process. Information about the exact build version is available as the `sys.subversion` variable, a 3-tuple of `(interpreter-name, branch-name, revision-range)`. For example, at the time of writing my copy of 2.5 was reporting ('CPython', 'trunk', '45313:45315').

  This information is also available to C extensions via the `Py_GetBuildInfo()` function that returns a string of build information like this: "trunk:45355:45356M, Apr 13 2006, 07:42:19". (Contributed by Barry Warsaw.)

- Another new function, `sys._current_frames()`, returns the current stack frames for all running threads as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called. (Contributed by Tim Peters.)

- The `TarFile` class in the `tarfile` module now has an `extractall()` method that extracts all members from the archive into the current working directory. It's also possible to set
a different directory as the extraction target, and to unpack only a subset of the archive's members.

The compression used for a tarfile opened in stream mode can now be autodetected using the mode `r|*'. (Contributed by Lars Gustäbel.)

- The `threading` module now lets you set the stack size used when new threads are created. The `stack_size([size])` function returns the currently configured stack size, and supplying the optional `size` parameter sets a new value. Not all platforms support changing the stack size, but Windows, POSIX threading, and OS/2 all do. (Contributed by Andrew MacIntyre.)

- The `unicodedata` module has been updated to use version 4.1.0 of the Unicode character database. Version 3.2.0 is required by some specifications, so it's still available as `unicodedata.ucd_3_2_0`.

- New module: the `uuid` module generates universally unique identifiers (UUIDs) according to RFC 4122. The RFC defines several different UUID versions that are generated from a starting string, from system properties, or purely randomly. This module contains a UUID class and functions named `uuid1()`, `uuid3()`, `uuid4()`, and `uuid5()` to generate different versions of UUID. (Version 2 UUIDs are not specified in RFC 4122 and are not supported by this module.)

```python
>>> import uuid
>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')
```
The `weakref` module's `WeakKeyDictionary` and `WeakValueDictionary` types gained new methods for iterating over the weak references contained in the dictionary. `iterkeyrefs()` and `keyrefs()` methods were added to `WeakKeyDictionary`, and `itervaluerefs()` and `valuerefs()` were added to `WeakValueDictionary`. (Contributed by Fred L. Drake, Jr.)

The `webbrowser` module received a number of enhancements. It's now usable as a script with `python -m webbrowser`, taking a URL as the argument; there are a number of switches to control the behaviour (`-n` for a new browser window, `-t` for a new tab). New module-level functions, `open_new()` and `open_new_tab()`, were added to support this. The module's `open()` function supports an additional feature, an `autoraise` parameter that signals whether to raise the open window when possible. A number of additional browsers were added to the supported list such as Firefox, Opera, Konqueror, and elinks. (Contributed by Oleg Broytmann and Georg Brandl.)
The `xmlrpc` module now supports returning `datetime` objects for the XML-RPC date type. Supply `use_datetime=True` to the `loads()` function or the Unmarshaller class to enable this feature. (Contributed by Skip Montanaro.)

The `zipfile` module now supports the ZIP64 version of the format, meaning that a .zip archive can now be larger than 4 GiB and can contain individual files larger than 4 GiB. (Contributed by Ronald Oussoren.)

The `zlib` module's Compress and Decompress objects now support a `copy()` method that makes a copy of the object's internal state and returns a new Compress or Decompress object. (Contributed by Chris AtLee.)

**The ctypes package**

The `ctypes` package, written by Thomas Heller, has been added to the standard library. `ctypes` lets you call arbitrary functions in shared libraries or DLLs. Long-time users may remember the `dl` module, which provides functions for loading shared libraries and calling functions in them. The `ctypes` package is much fancier.

To load a shared library or DLL, you must create an instance of the `CDLL` class and provide the name or path of the shared library or DLL. Once that's done, you can call arbitrary functions by accessing them as attributes of the `CDLL` object.

```python
import ctypes
```
```python
c = libc = ctypes.CDLL('libc.so.6')
result = libc.printf("Line of output\n")
```

Type constructors for the various C types are provided: `c_int()`, `c_float()`, `c_double()`, `c_char_p()` (equivalent to `char *`), and so forth. Unlike Python’s types, the C versions are all mutable; you can assign to their value attribute to change the wrapped value. Python integers and strings will be automatically converted to the corresponding C types, but for other types you must call the correct type constructor. (And I mean must; getting it wrong will often result in the interpreter crashing with a segmentation fault.)

You shouldn’t use `c_char_p()` with a Python string when the C function will be modifying the memory area, because Python strings are supposed to be immutable; breaking this rule will cause puzzling bugs. When you need a modifiable memory area, use `create_string_buffer()`:

```python
s = "this is a string"
buf = ctypes.create_string_buffer(s)
libc.strfry(buf)
```

C functions are assumed to return integers, but you can set the restype attribute of the function object to change this:

```python
>>> libc.atof('2.71828')
-1783957616
>>> libc.atof.restype = ctypes.c_double
>>> libc.atof('2.71828')
2.71828
```

`ctypes` also provides a wrapper for Python’s C API as the `ctypes.pythonapi` object. This object does not release the global
interpreter lock before calling a function, because the lock must be held when calling into the interpreter's code. There's a `py_object()` type constructor that will create a `PyObject *` pointer. A simple usage:

```python
import ctypes

d = {}
c ctypes.pythonapi.PyObject_SetItem(ctypes.py_object(ctypes.py_object("abc")), ctypes.py_object(1))
```

Don't forget to use `py_object()`; if it's omitted you end up with a segmentation fault.

`ctypes` has been around for a while, but people still write and distribution hand-coded extension modules because you can't rely on `ctypes` being present. Perhaps developers will begin to write Python wrappers atop a library accessed through `ctypes` instead of extension modules, now that `ctypes` is included with core Python.

**See also:**

http://starship.python.net/crew/theller/ctypes/

The ctypes web page, with a tutorial, reference, and FAQ.

The documentation for the `ctypes` module.

**The ElementTree package**

A subset of Fredrik Lundh's ElementTree library for processing XML has been added to the standard library as `xml.etree`. The available
modules are ElementTree, ElementPath, and ElementInclude from ElementTree 1.2.6. The cElementTree accelerator module is also included.

The rest of this section will provide a brief overview of using ElementTree. Full documentation for ElementTree is available at http://effbot.org/zone/element-index.htm.

ElementTree represents an XML document as a tree of element nodes. The text content of the document is stored as the text and tail attributes of (This is one of the major differences between ElementTree and the Document Object Model; in the DOM there are many different types of node, including TextNode.)

The most commonly used parsing function is parse(), that takes either a string (assumed to contain a filename) or a file-like object and returns an ElementTree instance:

```python
from xml.etree import ElementTree as ET

tree = ET.parse('ex-1.xml')

feed = urllib.urlopen('http://planet.python.org/rss10.xml')
tree = ET.parse(feed)
```

Once you have an ElementTree instance, you can call its getroot() method to get the root Element node.

There's also an XML() function that takes a string literal and returns an Element node (not an ElementTree). This function provides a tidy way to incorporate XML fragments, approaching the convenience of an XML literal:
Each XML element supports some dictionary-like and some list-like access methods. Dictionary-like operations are used to access attribute values, and list-like operations are used to access child nodes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>elem[n]</code></td>
<td>Returns n'th child element.</td>
</tr>
<tr>
<td><code>elem[m:n]</code></td>
<td>Returns list of m'th through n'th child elements.</td>
</tr>
<tr>
<td><code>len(elem)</code></td>
<td>Returns number of child elements.</td>
</tr>
<tr>
<td><code>list(elem)</code></td>
<td>Returns list of child elements.</td>
</tr>
<tr>
<td><code>elem.append(elem2)</code></td>
<td>Adds <code>elem2</code> as a child.</td>
</tr>
<tr>
<td><code>elem.insert(index, elem)</code></td>
<td>Inserts <code>elem</code> at the specified location.</td>
</tr>
<tr>
<td><code>del elem[n]</code></td>
<td>Deletes n'th child element.</td>
</tr>
<tr>
<td><code>elem.keys()</code></td>
<td>Returns list of attribute names.</td>
</tr>
<tr>
<td><code>elem.get(name)</code></td>
<td>Returns value of attribute <code>name</code>.</td>
</tr>
<tr>
<td><code>elem.set(name, value)</code></td>
<td>Sets new value for attribute <code>name</code>.</td>
</tr>
<tr>
<td><code>elem.attrib</code></td>
<td>Retrieves the dictionary containing attributes.</td>
</tr>
<tr>
<td><code>del elem.attrib[name]</code></td>
<td>Deletes attribute <code>name</code>.</td>
</tr>
</tbody>
</table>
Comments and processing instructions are also represented as Element nodes. To check if a node is a comment or processing instructions:

```python
if elem.tag is ET.Comment:
    ...
elif elem.tag is ET.ProcessingInstruction:
    ...
```

To generate XML output, you should call the ElementTree.write() method. Like parse(), it can take either a string or a file-like object:

```python
# Encoding is US-ASCII
tree.write('output.xml')

# Encoding is UTF-8
f = open('output.xml', 'w')
tree.write(f, encoding='utf-8')
```

(Caution: the default encoding used for output is ASCII. For general XML work, where an element's name may contain arbitrary Unicode characters, ASCII isn't a very useful encoding because it will raise an exception if an element's name contains any characters with values greater than 127. Therefore, it's best to specify a different encoding such as UTF-8 that can handle any Unicode character.)

This section is only a partial description of the ElementTree interfaces. Please read the package's official documentation for more details.

See also:

http://effbot.org/zone/element-index.htm

Official documentation for ElementTree.
The hashlib package

A new `hashlib` module, written by Gregory P. Smith, has been added to replace the md5 and sha modules. `hashlib` adds support for additional secure hashes (SHA-224, SHA-256, SHA-384, and SHA-512). When available, the module uses OpenSSL for fast platform optimized implementations of algorithms.

The old md5 and sha modules still exist as wrappers around hashlib to preserve backwards compatibility. The new module's interface is very close to that of the old modules, but not identical. The most significant difference is that the constructor functions for creating new hashing objects are named differently.

```
# Old versions
h = md5.md5()
h = md5.new()

# New version
h = hashlib.md5()

# Old versions
h = sha.sha()
h = sha.new()

# New version
h = hashlib.sha1()

# Hash that weren't previously available
h = hashlib.sha224()
h = hashlib.sha256()
h = hashlib.sha384()
h = hashlib.sha512()
```
Once a hash object has been created, its methods are the same as before: `update(string)` hashes the specified string into the current digest state, `digest()` and `hexdigest()` return the digest value as a binary string or a string of hex digits, and `copy()` returns a new hashing object with the same digest state.

See also: The documentation for the `hashlib` module.

The sqlite3 package

The `pysqlite` module ([http://www.pysqlite.org](http://www.pysqlite.org)), a wrapper for the SQLite embedded database, has been added to the standard library under the package name `sqlite3`.

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

`pysqlite` was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by [PEP 249](http://www.python.org/dev/peps/pep-0249/).

If you're compiling the Python source yourself, note that the source tree doesn't include the SQLite code, only the wrapper module. You'll need to have the SQLite libraries and headers installed before compiling Python, and the build process will compile the module when the necessary headers are available.
To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the /tmp/example file:

```python
conn = sqlite3.connect('/tmp/example')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its `execute()` method to perform SQL commands:

```python
# Create table
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')

# Insert a row of data
c.execute("""insert into stocks
values ('2006-01-05','BUY','RHAT',100,35.14)""")
```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python’s string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API’s parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor’s `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
          ]:
c.execute('insert into stocks values (?,?,?,?,?)

To retrieve data after executing a SELECT statement, you can either
treat the cursor as an iterator, call the cursor's fetchone() method
to retrieve a single matching row, or call fetchall() to get a list of
the matching rows.

This example uses the iterator form:

```python
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...     print row
...
(2006-01-05, 'BUY', 'RHAT', 100, 35.14)
(2006-03-28, 'BUY', 'IBM', 1000, 45.0)
(2006-04-06, 'SELL', 'IBM', 500, 53.0)
(2006-04-05, 'BUY', 'MSOFT', 1000, 72.0)
```

For more information about the SQL dialect supported by SQLite, see
See also:

**http://www.pysqlite.org**  
The pysqlite web page.

**https://www.sqlite.org**  
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

The documentation for the `sqlite3` module.

**PEP 249 - Database API Specification 2.0**  
PEP written by Marc-André Lemburg.

---

**The wsgiref package**

The Web Server Gateway Interface (WSGI) v1.0 defines a standard interface between web servers and Python web applications and is described in **PEP 333**. The `wsgiref` package is a reference implementation of the WSGI specification.

The package includes a basic HTTP server that will run a WSGI application; this server is useful for debugging but isn’t intended for production use. Setting up a server takes only a few lines of code:

```python
from wsgiref import simple_server

wsgi_app = ...

host = ''
port = 8000
httpd = simple_server.make_server(host, port, wsgi_app)
```

```
See also:

**http://www.wsgi.org**
A central web site for WSGI-related resources.

**PEP 333 - Python Web Server Gateway Interface v1.0**
PEP written by Phillip J. Eby.
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The Python source tree was converted from CVS to Subversion, in a complex migration procedure that was supervised and flawlessly carried out by Martin von Löwis. The procedure was developed as PEP 347.

- Coverity, a company that markets a source code analysis tool called Prevent, provided the results of their examination of the Python source code. The analysis found about 60 bugs that were quickly fixed. Many of the bugs were refcounting problems, often occurring in error-handling code. See https://scan.coverity.com for the statistics.

- The largest change to the C API came from PEP 353, which modifies the interpreter to use a Py_ssize_t type definition instead of int. See the earlier section PEP 353: Using ssize_t as the index type for a discussion of this change.

- The design of the bytecode compiler has changed a great deal, no longer generating bytecode by traversing the parse tree. Instead the parse tree is converted to an abstract syntax tree (or AST), and it is the abstract syntax tree that’s traversed to produce the bytecode.

It’s possible for Python code to obtain AST objects by using the compile() built-in and specifying _ast.PyCF_ONLY_AST as the value of the flags parameter:

```python
from _ast import PyCF_ONLY_AST
```
```python
ast = compile("""a=0
for i in range(10):
a += i
""", "<string>", 'exec', PyCF_ONLY_AST)

assignment = ast.body[0]
for_loop = ast.body[1]
```

No official documentation has been written for the AST code yet, but PEP 339 discusses the design. To start learning about the code, read the definition of the various AST nodes in Parser/Python.asdl. A Python script reads this file and generates a set of C structure definitions in Include/Python-ast.h. The PyParser_ASTFromString() and PyParser_ASTFromFile(), defined in Include/pythonrun.h, take Python source as input and return the root of an AST representing the contents. This AST can then be turned into a code object by PyAST_Compile(). For more information, read the source code, and then ask questions on python-dev.

The AST code was developed under Jeremy Hylton's management, and implemented by (in alphabetical order) Brett Cannon, Nick Coghlan, Grant Edwards, John Ehresman, Kurt Kaiser, Neal Norwitz, Tim Peters, Armin Rigo, and Neil Schemenauer, plus the participants in a number of AST sprints at conferences such as PyCon.

- Evan Jones's patch to obmalloc, first described in a talk at PyCon DC 2005, was applied. Python 2.4 allocated small objects in 256K-sized arenas, but never freed arenas. With this patch, Python will free arenas when they're empty. The net effect is that on some platforms, when you allocate many objects, Python's
memory usage may actually drop when you delete them and the memory may be returned to the operating system. (Implemented by Evan Jones, and reworked by Tim Peters.)

Note that this change means extension modules must be more careful when allocating memory. Python’s API has many different functions for allocating memory that are grouped into families. For example, `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()` are one family that allocates raw memory, while `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()` are another family that's supposed to be used for creating Python objects.

Previously these different families all reduced to the platform's malloc() and free() functions. This meant it didn't matter if you got things wrong and allocated memory with the PyMem() function but freed it with the PyObject() function. With 2.5's changes to obmalloc, these families now do different things and mismatches will probably result in a segfault. You should carefully test your C extension modules with Python 2.5.

- The built-in set types now have an official C API. Call `PySet_New()` and `PyFrozenSet_New()` to create a new set, `PySet_Add()` and `PySet_Discard()` to add and remove elements, and `PySet_Contains()` and `PySet_Size()` to examine the set's state. (Contributed by Raymond Hettinger.)

- C code can now obtain information about the exact revision of the Python interpreter by calling the `Py_GetBuildInfo()` function that returns a string of build information like this: "trunk:45355:45356M, Apr 13 2006, 07:42:19". (Contributed by Barry Warsaw.)
- Two new macros can be used to indicate C functions that are local to the current file so that a faster calling convention can be used. `Py_LOCAL(type)` declares the function as returning a value of the specified type and uses a fast-calling qualifier. `Py_LOCAL_INLINE(type)` does the same thing and also requests the function be inlined. If `PY_LOCAL_AGGRESSIVE()` is defined before `python.h` is included, a set of more aggressive optimizations are enabled for the module; you should benchmark the results to find out if these optimizations actually make the code faster. (Contributed by Fredrik Lundh at the NeedForSpeed sprint.)

- `PyErr_NewException(name, base, dict)` can now accept a tuple of base classes as its base argument. (Contributed by Georg Brandl.)

- The `PyErrWarn()` function for issuing warnings is now deprecated in favour of `PyErr_WarnEx(category, message, stacklevel)` which lets you specify the number of stack frames separating this function and the caller. A `stacklevel` of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth. (Added by Neal Norwitz.)

- The CPython interpreter is still written in C, but the code can now be compiled with a C++ compiler without errors. (Implemented by Anthony Baxter, Martin von Löwis, Skip Montanaro.)

- The `PyRange_New()` function was removed. It was never documented, never used in the core code, and had dangerously lax error checking. In the unlikely case that your extensions were using it, you can replace it by something like the following:
Port-Specific Changes

- MacOS X (10.3 and higher): dynamic loading of modules now uses the dlopen() function instead of MacOS-specific functions.
- MacOS X: an --enable-universal-sdk switch was added to the configure script that compiles the interpreter as a universal binary able to run on both PowerPC and Intel processors. (Contributed by Ronald Oussoren; bpo-2573.)
- Windows: .dll is no longer supported as a filename extension for extension modules. .pyd is now the only filename extension that will be searched for.
Porting to Python 2.5

This section lists previously described changes that may require changes to your code:

- ASCII is now the default encoding for modules. It’s now a syntax error if a module contains string literals with 8-bit characters but doesn’t have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error.
- Previously, the gi_frame attribute of a generator was always a frame object. Because of the PEP 342 changes described in section PEP 342: New Generator Features, it’s now possible for gi_frame to be None.
- A new warning, UnicodeWarning, is triggered when you attempt to compare a Unicode string and an 8-bit string that can’t be converted to Unicode using the default ASCII encoding. Previously such comparisons would raise a UnicodeDecodeError exception.
- Library: the csv module is now stricter about multi-line quoted fields. If your files contain newlines embedded within fields, the input should be split into lines in a manner which preserves the newline characters.
- Library: the locale module’s format() function’s would previously accept any string as long as no more than one %char specifier appeared. In Python 2.5, the argument must be exactly one %char specifier with no surrounding text.
- Library: The pickle and cPickle modules no longer accept a return value of None from the __reduce__() method; the method must return a tuple of arguments instead. The modules also no longer accept the deprecated bin keyword parameter.
- Library: The SimpleXMLRPCServer and DocXMLRPCServer classes now have a rpc_paths attribute that constrains XML-
RPC operations to a limited set of URL paths; the default is to allow only '//' and '/RPC2'. Setting rpc_paths to None or an empty tuple disables this path checking.

- C API: Many functions now use Py_ssize_t instead of int to allow processing more data on 64-bit machines. Extension code may need to make the same change to avoid warnings and to support 64-bit machines. See the earlier section PEP 353: Using ssize_t as the index type for a discussion of this change.

- C API: The obmalloc changes mean that you must be careful to not mix usage of the PyMem_*() and PyObject_*() families of functions. Memory allocated with one family’s *_Malloc() must be freed with the corresponding family’s *_Free() function.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Nick Coghlan, Phillip J. Eby, Lars Gustäbel, Raymond Hettinger, Ralf W. Grosse-Kunstleve, Kent Johnson, Iain Lowe, Martin von Löwis, Fredrik Lundh, Andrew McNamara, Skip Montanaro, Gustavo Niemeyer, Paul Prescod, James Pryor, Mike Rovner, Scott Weikart, Barry Warsaw, Thomas Wouters.
What’s New in Python 2.4

Author: A.M. Kuchling

This article explains the new features in Python 2.4.1, released on March 30, 2005.

Python 2.4 is a medium-sized release. It doesn’t introduce as many changes as the radical Python 2.2, but introduces more features than the conservative 2.3 release. The most significant new language features are function decorators and generator expressions; most other changes are to the standard library.

According to the CVS change logs, there were 481 patches applied and 502 bugs fixed between Python 2.3 and 2.4. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of every single new feature, but instead provides a brief introduction to each feature. For full details, you should refer to the documentation for Python 2.4, such as the Python Library Reference and the Python Reference Manual. Often you will be referred to the PEP for a particular new feature for explanations of the implementation and design rationale.
PEP 218: Built-In Set Objects

Python 2.3 introduced the sets module. C implementations of set data types have now been added to the Python core as two new built-in types, `set(iterable)` and `frozenset(iterable)`. They provide high speed operations for membership testing, for eliminating duplicates from sequences, and for mathematical operations like unions, intersections, differences, and symmetric differences.

```python
>>> a = set('abracadabra')      # form a set
>>> 'z' in a                   # fast membership testing
False
>>> a                          # unique letters
set(['a', 'r', 'b', 'c', 'd'])
>>> ''.join(a)                 # convert back into a string
'arbcd'

>>> b = set('alacazam')        # form another set
>>> a - b                      # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                      # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                      # letters in both a and b
set(['a', 'c'])
>>> a ^ b                      # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])

>>> a.add('z')                 # add a new element
>>> a                          # add multiple elements
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'x', 'z'])

>>> a.remove('x')              # take one element out
>>> a                          # take one element out
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'z'])
```
The `frozenset()` type is an immutable version of `set()`. Since it is immutable and hashable, it may be used as a dictionary key or as a member of another set.

The `sets` module remains in the standard library, and may be useful if you wish to subclass the `Set` or `ImmutableSet` classes. There are currently no plans to deprecate the module.

**See also:**

**PEP 218 - Adding a Built-In Set Object Type**

Originally proposed by Greg Wilson and ultimately implemented by Raymond Hettinger.
PEP 237: Unifying Long Integers and Integers

The lengthy transition process for this PEP, begun in Python 2.2, takes another step forward in Python 2.4. In 2.3, certain integer operations that would behave differently after int/long unification triggered `FutureWarning` warnings and returned values limited to 32 or 64 bits (depending on your platform). In 2.4, these expressions no longer produce a warning and instead produce a different result that's usually a long integer.

The problematic expressions are primarily left shifts and lengthy hexadecimal and octal constants. For example, `2 << 32` results in a warning in 2.3, evaluating to 0 on 32-bit platforms. In Python 2.4, this expression now returns the correct answer, 8589934592.

See also:

**PEP 237 - Unifying Long Integers and Integers**

Original PEP written by Moshe Zadka and GvR. The changes for 2.4 were implemented by Kalle Svensson.
PEP 289: Generator Expressions

The iterator feature introduced in Python 2.2 and the `itertools` module make it easier to write programs that loop through large data sets without having the entire data set in memory at one time. List comprehensions don't fit into this picture very well because they produce a Python list object containing all of the items. This unavoidably pulls all of the objects into memory, which can be a problem if your data set is very large. When trying to write a functionally-styled program, it would be natural to write something like:

```python
links = [link for link in get_all_links() if not link
for link in links:
    ...
```

instead of

```python
for link in get_all_links():
    if link.followed:
        continue
    ...
```

The first form is more concise and perhaps more readable, but if you're dealing with a large number of link objects you'd have to write the second form to avoid having all link objects in memory at the same time.

Generator expressions work similarly to list comprehensions but don't materialize the entire list; instead they create a generator that will return elements one by one. The above example could be written as:

```python
links = (link for link in get_all_links() if not link
```
Generator expressions always have to be written inside parentheses, as in the above example. The parentheses signalling a function call also count, so if you want to create an iterator that will be immediately passed to a function you could write:

```python
print sum(obj.count for obj in list_all_objects())
```

Generator expressions differ from list comprehensions in various small ways. Most notably, the loop variable (\texttt{obj} in the above example) is not accessible outside of the generator expression. List comprehensions leave the variable assigned to its last value; future versions of Python will change this, making list comprehensions match generator expressions in this respect.

**See also:**

**PEP 289 - Generator Expressions**

Proposed by Raymond Hettinger and implemented by Jiwon Seo with early efforts steered by Hye-Shik Chang.
PEP 292: Simpler String Substitutions

Some new classes in the standard library provide an alternative mechanism for substituting variables into strings; this style of substitution may be better for applications where untrained users need to edit templates.

The usual way of substituting variables by name is the % operator:

```python
>>> '%(page)i: %(title)s' % {'page': 2, 'title': 'The Best of Times'}
2: The Best of Times
```

When writing the template string, it can be easy to forget the i or s after the closing parenthesis. This isn't a big problem if the template is in a Python module, because you run the code, get an “Unsupported format character” ValueError, and fix the problem. However, consider an application such as Mailman where template strings or translations are being edited by users who aren't aware of the Python language. The format string's syntax is complicated to explain to such users, and if they make a mistake, it's difficult to provide helpful feedback to them.

PEP 292 adds a Template class to the string module that uses $ to indicate a substitution:

```python
>>> import string
>>> t = string.Template('${page: $title}')
>>> t.substitute({'page': 2, 'title': 'The Best of Times'})
2: The Best of Times
```

If a key is missing from the dictionary, the substitute() method will
raise a **KeyError**. There's also a `safe_substitute()` method that ignores missing keys:

```python
>>> t = string.Template('{$page: $title}')
>>> t.safe_substitute({'page': 3})
'3: $title'
```

See also:

**PEP 292 - Simpler String Substitutions**

Written and implemented by Barry Warsaw.
PEP 318: Decorators for Functions and Methods

Python 2.2 extended Python’s object model by adding static methods and class methods, but it didn’t extend Python’s syntax to provide any new way of defining static or class methods. Instead, you had to write a `def` statement in the usual way, and pass the resulting method to a `staticmethod()` or `classmethod()` function that would wrap up the function as a method of the new type. Your code would look like this:

```python
class C:
    def meth (cls):
        ...

    meth = classmethod(meth)  # Rebind name to wrapped-up class method
```

If the method was very long, it would be easy to miss or forget the `classmethod()` invocation after the function body.

The intention was always to add some syntax to make such definitions more readable, but at the time of 2.2’s release a good syntax was not obvious. Today a good syntax still isn’t obvious but users are asking for easier access to the feature; a new syntactic feature has been added to meet this need.

The new feature is called “function decorators”. The name comes from the idea that `classmethod()`, `staticmethod()`, and friends are storing additional information on a function object; they’re decorating functions with more details.
The notation borrows from Java and uses the '@' character as an indicator. Using the new syntax, the example above would be written:

```python
class C:
    @classmethod
    def meth(cls):
        ...
```

The `@classmethod` is shorthand for the `meth=classmethod(meth)` assignment. More generally, if you have the following:

```python
@A
@B
@C
def f():
    ...
```

It's equivalent to the following pre-decorator code:

```python
def f(): ...
f = A(B(C(f)))
```

Decorators must come on the line before a function definition, one decorator per line, and can't be on the same line as the `def` statement, meaning that `@A def f(): ...` is illegal. You can only decorate function definitions, either at the module level or inside a class; you can't decorate class definitions.

A decorator is just a function that takes the function to be decorated as an argument and returns either the same function or some new object. The return value of the decorator need not be callable (though it typically is), unless further decorators will be applied to the result. It's
easy to write your own decorators. The following simple example just sets an attribute on the function object:

```python
>>> def deco(func):
...     func.attr = 'decorated'
...     return func
...
>>> @deco
... def f():
...     pass
...
>>> f
<function f at 0x402ef0d4>
>>> f.attr
'decorated'
>>> 
```

As a slightly more realistic example, the following decorator checks that the supplied argument is an integer:

```python
def require_int(func):
    def wrapper(arg):
        assert isinstance(arg, int)
        return func(arg)
    return wrapper

@require_int
def p1(arg):
    print arg

@require_int
def p2(arg):
    print arg*2

An example in PEP 318 contains a fancier version of this idea that lets you both specify the required type and check the returned type.
Decorator functions can take arguments. If arguments are supplied, your decorator function is called with only those arguments and must return a new decorator function; this function must take a single function and return a function, as previously described. In other words, \@A \@B \@C(args) becomes:

```python
def f(): ...
_deco = C(args)
f = A(B(_deco(f)))
```

Getting this right can be slightly brain-bending, but it’s not too difficult.

A small related change makes the `func_name` attribute of functions writable. This attribute is used to display function names in tracebacks, so decorators should change the name of any new function that’s constructed and returned.

See also:

**PEP 318 - Decorators for Functions, Methods and Classes**

Written by Kevin D. Smith, Jim Jewett, and Skip Montanaro. Several people wrote patches implementing function decorators, but the one that was actually checked in was patch #979728, written by Mark Russell.

[https://wiki.python.org/moin/PythonDecoratorLibrary](https://wiki.python.org/moin/PythonDecoratorLibrary)

This Wiki page contains several examples of decorators.
PEP 322: Reverse Iteration

A new built-in function, `reversed(seq)`, takes a sequence and returns an iterator that loops over the elements of the sequence in reverse order.

```python
>>> for i in reversed(xrange(1,4)):
...     print i
...
3
2
1
```

Compared to extended slicing, such as `range(1,4)[::-1]`, `reversed()` is easier to read, runs faster, and uses substantially less memory.

Note that `reversed()` only accepts sequences, not arbitrary iterators. If you want to reverse an iterator, first convert it to a list with `list()`.

```python
>>> input = open('/etc/passwd', 'r')
>>> for line in reversed(list(input)):
...     print line
...
root:*:0:0:System Administrator:/var/root:/bin/tcsh
```

See also:

**PEP 322 - Reverse Iteration**

Written and implemented by Raymond Hettinger.
PEP 324: New subprocess Module

The standard library provides a number of ways to execute a subprocess, offering different features and different levels of complexity. `os.system(command)` is easy to use, but slow (it runs a shell process which executes the command) and dangerous (you have to be careful about escaping the shell's metacharacters). The popen2 module offers classes that can capture standard output and standard error from the subprocess, but the naming is confusing. The `subprocess` module cleans this up, providing a unified interface that offers all the features you might need.

Instead of popen2’s collection of classes, `subprocess` contains a single class called Popen whose constructor supports a number of different keyword arguments.

```python
class Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0):
```

`args` is commonly a sequence of strings that will be the arguments to the program executed as the subprocess. (If the `shell` argument is true, `args` can be a string which will then be passed on to the shell for interpretation, just as `os.system()` does.)

`stdin`, `stdout`, and `stderr` specify what the subprocess’s input, output, and error streams will be. You can provide a file object or a file descriptor, or you can use the constant `subprocess.PIPE` to create a pipe between the subprocess and the parent.
The constructor has a number of handy options:

- `close_fds` requests that all file descriptors be closed before running the subprocess.
- `cwd` specifies the working directory in which the subprocess will be executed (defaulting to whatever the parent's working directory is).
- `env` is a dictionary specifying environment variables.
- `preexec_fn` is a function that gets called before the child is started.
- `universal_newlines` opens the child's input and output using Python's `universal newlines` feature.

Once you've created the `Popen` instance, you can call its `wait()` method to pause until the subprocess has exited, `poll()` to check if it's exited without pausing, or `communicate(data)` to send the string `data` to the subprocess's standard input. `communicate(data)` then reads any data that the subprocess has sent to its standard output or standard error, returning a tuple `(stdout_data, stderr_data)`.

call() is a shortcut that passes its arguments along to the `Popen` constructor, waits for the command to complete, and returns the status code of the subprocess. It can serve as a safer analog to `os.system()`:

```python
sts = subprocess.call(['dpkg', '-i', '/tmp/new-package.deb'])
if sts == 0:
    # Success
    ...
else:
    # dpkg returned an error
    ...
```
The command is invoked without use of the shell. If you really do want to use the shell, you can add `shell=True` as a keyword argument and provide a string instead of a sequence:

```python
sts = subprocess.call('dpkg -i /tmp/new-package.deb', shell=True)
```

The PEP takes various examples of shell and Python code and shows how they’d be translated into Python code that uses `subprocess`. Reading this section of the PEP is highly recommended.

**See also:**

**PEP 324** - subprocess - New process module

Written and implemented by Peter Åstrand, with assistance from Fredrik Lundh and others.
PEP 327: Decimal Data Type

Python has always supported floating-point (FP) numbers, based on the underlying C double type, as a data type. However, while most programming languages provide a floating-point type, many people (even programmers) are unaware that floating-point numbers don’t represent certain decimal fractions accurately. The new Decimal type can represent these fractions accurately, up to a user-specified precision limit.

Why is Decimal needed?

The limitations arise from the representation used for floating-point numbers. FP numbers are made up of three components:

- The sign, which is positive or negative.
- The mantissa, which is a single-digit binary number followed by a fractional part. For example, 1.01 in base-2 notation is $1 + 0/2 + 1/4$, or 1.25 in decimal notation.
- The exponent, which tells where the decimal point is located in the number represented.

For example, the number 1.25 has positive sign, a mantissa value of 1.01 (in binary), and an exponent of 0 (the decimal point doesn’t need to be shifted). The number 5 has the same sign and mantissa, but the exponent is 2 because the mantissa is multiplied by 4 (2 to the power of the exponent 2); $1.25 \times 4$ equals 5.

Modern systems usually provide floating-point support that conforms to a standard called IEEE 754. C’s double type is usually implemented as a 64-bit IEEE 754 number, which uses 52 bits of space for the mantissa. This means that numbers can only be
specified to 52 bits of precision. If you’re trying to represent numbers whose expansion repeats endlessly, the expansion is cut off after 52 bits. Unfortunately, most software needs to produce output in base 10, and common fractions in base 10 are often repeating decimals in binary. For example, 1.1 decimal is binary 1.0001100110011...; .1 = 1/16 + 1/32 + 1/256 plus an infinite number of additional terms. IEEE 754 has to chop off that infinitely repeated decimal after 52 digits, so the representation is slightly inaccurate.

Sometimes you can see this inaccuracy when the number is printed:

```plaintext
>>> 1.1
1.1000000000000001
```

The inaccuracy isn’t always visible when you print the number because the FP-to-decimal-string conversion is provided by the C library, and most C libraries try to produce sensible output. Even if it’s not displayed, however, the inaccuracy is still there and subsequent operations can magnify the error.

For many applications this doesn’t matter. If I’m plotting points and displaying them on my monitor, the difference between 1.1 and 1.1000000000000001 is too small to be visible. Reports often limit output to a certain number of decimal places, and if you round the number to two or three or even eight decimal places, the error is never apparent. However, for applications where it does matter, it’s a lot of work to implement your own custom arithmetic routines.

Hence, the Decimal type was created.

The Decimal type
A new module, `decimal`, was added to Python’s standard library. It contains two classes, `Decimal` and `Context`. `Decimal` instances represent numbers, and `Context` instances are used to wrap up various settings such as the precision and default rounding mode.

`Decimal` instances are immutable, like regular Python integers and FP numbers; once it’s been created, you can’t change the value an instance represents. `Decimal` instances can be created from integers or strings:

```python
>>> import decimal
>>> decimal.Decimal(1972)
Decimal("1972")
>>> decimal.Decimal("1.1")
Decimal("1.1")
```

You can also provide tuples containing the sign, the mantissa represented as a tuple of decimal digits, and the exponent:

```python
>>> decimal.Decimal((1, (1, 4, 7, 5), -2))
Decimal("-14.75")
```

Cautionary note: the sign bit is a Boolean value, so 0 is positive and 1 is negative.

Converting from floating-point numbers poses a bit of a problem: should the FP number representing 1.1 turn into the decimal number for exactly 1.1, or for 1.1 plus whatever inaccuracies are introduced? The decision was to dodge the issue and leave such a conversion out of the API. Instead, you should convert the floating-point number into a string using the desired precision and pass the string to the `Decimal` constructor:
Once you have Decimal instances, you can perform the usual mathematical operations on them. One limitation: exponentiation requires an integer exponent:

```python
>>> a = decimal.Decimal('35.72')
>>> b = decimal.Decimal('1.73')
>>> a+b
Decimal("37.45")
>>> a-b
Decimal("33.99")
>>> a*b
Decimal("61.7956")
>>> a/b
Decimal("20.64739884393063583815028902")
>>> a ** 2
Decimal("1275.9184")
>>> a**b
Traceback (most recent call last):
  ...  
  decimal.InvalidOperation: x ** (non-integer)
```

You can combine Decimal instances with integers, but not with floating-point numbers:

```python
>>> a + 4
Decimal("39.72")
>>> a + 4.5
Traceback (most recent call last):
  ...
```
Decimal numbers can be used with the `math` and `cmath` modules, but note that they'll be immediately converted to floating-point numbers before the operation is performed, resulting in a possible loss of precision and accuracy. You'll also get back a regular floating-point number and not a Decimal.

```python
>>> import math, cmath
>>> d = decimal.Decimal('123456789012.345')
>>> math.sqrt(d)
351364.18288201344
>>> cmath.sqrt(-d)
351364.18288201344j
```

Decimal instances have a `sqrt()` method that returns a Decimal, but if you need other things such as trigonometric functions you'll have to implement them.

```python
>>> d.sqrt()
Decimal("351364.1828820134592177245001")
```

### The Context type

Instances of the Context class encapsulate several settings for decimal operations:

- `prec` is the precision, the number of decimal places.
- `rounding` specifies the rounding mode. The `decimal` module has constants for the various possibilities: `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_HALF_EVEN`, and various others.
traps is a dictionary specifying what happens on encountering certain error conditions: either an exception is raised or a value is returned. Some examples of error conditions are division by zero, loss of precision, and overflow.

There's a thread-local default context available by calling getcontext(); you can change the properties of this context to alter the default precision, rounding, or trap handling. The following example shows the effect of changing the precision of the default context:

```python
>>> decimal.getcontext().prec = 28
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.1428571428571428571428571429")
>>> decimal.getcontext().prec = 9
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857143")
```

The default action for error conditions is selectable; the module can either return a special value such as infinity or not-a-number, or exceptions can be raised:

```python
>>> decimal.Decimal(1) / decimal.Decimal(0)
Traceback (most recent call last):
  ... 
decimal.DivisionByZero: x / 0
>>> decimal.getcontext().traps[decimal.DivisionByZero] = False
>>> decimal.Decimal(1) / decimal.Decimal(0)
Decimal("Infinity")
```

The Context instance also has various methods for formatting numbers such as to_eng_string() and to_sci_string().
For more information, see the documentation for the `decimal` module, which includes a quick-start tutorial and a reference.

See also:

**PEP 327 - Decimal Data Type**
Written by Facundo Batista and implemented by Facundo Batista, Eric Price, Raymond Hettinger, Aahz, and Tim Peters.

[http://www.lahey.com/float.htm](http://www.lahey.com/float.htm)
The article uses Fortran code to illustrate many of the problems that floating-point inaccuracy can cause.

A description of a decimal-based representation. This representation is being proposed as a standard, and underlies the new Python decimal type. Much of this material was written by Mike Cowlishaw, designer of the Rexx language.
PEP 328: Multi-line Imports

One language change is a small syntactic tweak aimed at making it easier to import many names from a module. In a `from module import names` statement, `names` is a sequence of names separated by commas. If the sequence is very long, you can either write multiple imports from the same module, or you can use backslashes to escape the line endings like this:

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer,
                        SimpleXMLRPCRequestHandler,
                        CGIXMLRPCRequestHandler,
                        resolve_dotted_attribute
```

The syntactic change in Python 2.4 simply allows putting the names within parentheses. Python ignores newlines within a parenthesized expression, so the backslashes are no longer needed:

```python
from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                SimpleXMLRPCRequestHandler,
                                CGIXMLRPCRequestHandler,
                                resolve_dotted_attribute)
```

The PEP also proposes that all `import` statements be absolute imports, with a leading `.` character to indicate a relative import. This part of the PEP was not implemented for Python 2.4, but was completed for Python 2.5.

See also:

**PEP 328 - Imports: Multi-Line and Absolute/Relative**

Written by Aahz. Multi-line imports were implemented by Dima
Dorfman.
PEP 331: Locale-Independent Float/String Conversions

The `locale` modules lets Python software select various conversions and display conventions that are localized to a particular country or language. However, the module was careful to not change the numeric locale because various functions in Python's implementation required that the numeric locale remain set to the 'C' locale. Often this was because the code was using the C library's `atof()` function.

Not setting the numeric locale caused trouble for extensions that used third-party C libraries, however, because they wouldn’t have the correct locale set. The motivating example was GTK+, whose user interface widgets weren’t displaying numbers in the current locale.

The solution described in the PEP is to add three new functions to the Python API that perform ASCII-only conversions, ignoring the locale setting:

- `PyOS_ascii_strtod(str, ptr)` and `PyOS_ascii_atof(str, ptr)` both convert a string to a C double.
- `PyOS_ascii_formatd(buffer, buf_len, format, d)` converts a double to an ASCII string.

The code for these functions came from the GLib library (https://developer.gnome.org/glib/stable/), whose developers kindly relicensed the relevant functions and donated them to the Python Software Foundation. The `locale` module can now change the numeric locale, letting extensions such as GTK+ produce the correct results.
See also:

**PEP 331** - Locale-Independent Float/String Conversions

Written by Christian R. Reis, and implemented by Gustavo Carneiro.
Other Language Changes

Here are all of the changes that Python 2.4 makes to the core Python language.

- Decorators for functions and methods were added ([PEP 318]).
- Built-in `set()` and `frozenset()` types were added ([PEP 218]). Other new built-ins include the `reversed(seq)` function ([PEP 322]).
- Generator expressions were added ([PEP 289]).
- Certain numeric expressions no longer return values restricted to 32 or 64 bits ([PEP 237]).
- You can now put parentheses around the list of names in a `from module import names` statement ([PEP 328]).
- The `dict.update()` method now accepts the same argument forms as the `dict` constructor. This includes any mapping, any iterable of key/value pairs, and keyword arguments. (Contributed by Raymond Hettinger.)
- The string methods `ljust()`, `rjust()`, and `center()` now take an optional argument for specifying a fill character other than a space. (Contributed by Raymond Hettinger.)
- Strings also gained an `rsplit()` method that works like the `split()` method but splits from the end of the string. (Contributed by Sean Reifschneider.)

```python
>>> 'www.python.org'.split('.', 1)
```
Three keyword parameters, `cmp`, `key`, and `reverse`, were added to the `sort()` method of lists. These parameters make some common usages of `sort()` simpler. All of these parameters are optional.

For the `cmp` parameter, the value should be a comparison function that takes two parameters and returns -1, 0, or +1 depending on how the parameters compare. This function will then be used to sort the list. Previously this was the only parameter that could be provided to `sort()`.

`key` should be a single-parameter function that takes a list element and returns a comparison key for the element. The list is then sorted using the comparison keys. The following example sorts a list case-insensitively:

```python
>>> L = ['A', 'b', 'c', 'D']
>>> L.sort()  # Case-sensitive sort
>>> L
['A', 'D', 'b', 'c']
>>> # Using 'key' parameter to sort list
>>> L.sort(key=lambda x: x.lower())
>>> L
['A', 'b', 'c', 'D']
>>> # Old-fashioned way
>>> L.sort(cmp=lambda x,y: cmp(x.lower(), y.lower))
>>> L
['A', 'b', 'c', 'D']
```

The last example, which uses the `cmp` parameter, is the old way
to perform a case-insensitive sort. It works but is slower than using a key parameter. Using key calls lower() method once for each element in the list while using cmp will call it twice for each comparison, so using key saves on invocations of the lower() method.

For simple key functions and comparison functions, it is often possible to avoid a lambda expression by using an unbound method instead. For example, the above case-insensitive sort is best written as:

```python
>>> L.sort(key=str.lower)
>>> L
['A', 'b', 'c', 'D']
```

Finally, the reverse parameter takes a Boolean value. If the value is true, the list will be sorted into reverse order. Instead of `L.sort(); L.reverse()`, you can now write `L.sort(reverse=True)`.

The results of sorting are now guaranteed to be stable. This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.

(All changes to sort() contributed by Raymond Hettinger.)

- There is a new built-in function `sorted(iterable)` that works like the in-place `list.sort()` method but can be used in expressions. The differences are:
- the input may be any iterable;
- a newly formed copy is sorted, leaving the original intact; and
- the expression returns the new sorted copy

```python
>>> L = [9, 7, 8, 3, 2, 4, 1, 6, 5]
>>> [10+i for i in sorted(L)]  # usable in a list comprehension
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> L  # original is left unchanged
[9, 7, 8, 3, 2, 4, 1, 6, 5]
>>> sorted('Monty Python')  # any iterable may be an input
[' ', 'M', 'P', 'h', 'n', 'o', 'o', 't', 'y']
>>> # List the contents of a dict sorted by key values
>>> colormap = dict(red=1, blue=2, green=3, black=4)
>>> for k, v in sorted(colormap.items()):
...     print k, v
...  black 4
  blue 2
  green 3
  red 1
  yellow 5
```

(Contributed by Raymond Hettinger.)

- Integer operations will no longer trigger an OverflowWarning. The OverflowWarning warning will disappear in Python 2.5.
- The interpreter gained a new switch, `-m`, that takes a name, searches for the corresponding module on `sys.path`, and runs the module as a script. For example, you can now run the Python profiler with `python -m profile`. (Contributed by Nick Coghlan.)
The `eval(expr, globals, locals)` and `execfile(filename, globals, locals)` functions and the `exec` statement now accept any mapping type for the `locals` parameter. Previously this had to be a regular Python dictionary. *(Contributed by Raymond Hettinger.)*

- The `zip()` built-in function and `itertools.izip()` now return an empty list if called with no arguments. Previously they raised a `TypeError` exception. This makes them more suitable for use with variable length argument lists:

  ```python
  def transpose(array):
      return zip(*array)
  ```

  ```python
  >>> transpose(((1, 2, 3), (4, 5, 6)))
  [(1, 4), (2, 5), (3, 6)]
  >>> transpose([])
  []
  ```

  *(Contributed by Raymond Hettinger.)*

- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`. The incomplete module object left behind would fool further imports of the same module into succeeding, leading to confusing errors. *(Fixed by Tim Peters.)*

- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error. *(Contributed by Raymond Hettinger.)*

**Optimizations**

- The inner loops for list and tuple slicing were optimized and now
run about one-third faster. The inner loops for dictionaries were also optimized, resulting in performance boosts for keys(), values(), items(), iterkeys(), itervalues(), and iteritems. (Contributed by Raymond Hettinger.)

- The machinery for growing and shrinking lists was optimized for speed and for space efficiency. Appending and popping from lists now runs faster due to more efficient code paths and less frequent use of the underlying system realloc(). List comprehensions also benefit. list.extend() was also optimized and no longer converts its argument into a temporary list before extending the base list. (Contributed by Raymond Hettinger.)
- list(), tuple(), map(), filter(), and zip() now run several times faster with non-sequence arguments that supply a __len__() method. (Contributed by Raymond Hettinger.)
- The methods list.__getitem__, dict.__getitem__, and dict.__contains__ are now implemented as method descriptor objects rather than wrapper descriptor objects. This form of access doubles their performance and makes them more suitable for use as arguments to functionals: map(mydict.__getitem__, keylist). (Contributed by Raymond Hettinger.)
- Added a new opcode, LIST_APPEND, that simplifies the generated bytecode for list comprehensions and speeds them up by about a third. (Contributed by Raymond Hettinger.)
- The peephole bytecode optimizer has been improved to produce shorter, faster bytecode; remarkably, the resulting bytecode is more readable. (Enhanced by Raymond Hettinger.)
- String concatenations in statements of the form s = s + "abc" and s += "abc" are now performed more efficiently in certain circumstances. This optimization won't be present in other Python implementations such as Jython, so you shouldn't rely on it; using the join() method of strings is still recommended when you want to efficiently glue a large number of strings together.
(Contributed by Armin Rigo.)

The net result of the 2.4 optimizations is that Python 2.4 runs the pystone benchmark around 5% faster than Python 2.3 and 35% faster than Python 2.2. (pystone is not a particularly good benchmark, but it's the most commonly used measurement of Python's performance. Your own applications may show greater or smaller benefits from Python 2.4.)
New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The asyncore module's `loop()` function now has a `count` parameter that lets you perform a limited number of passes through the polling loop. The default is still to loop forever.

- The base64 module now has more complete RFC 3548 support for Base64, Base32, and Base16 encoding and decoding, including optional case folding and optional alternative alphabets. (Contributed by Barry Warsaw.)

- The bisect module now has an underlying C implementation for improved performance. (Contributed by Dmitry Vasiliev.)

- The CJKCodecs collections of East Asian codecs, maintained by Hye-Shik Chang, was integrated into 2.4. The new encodings are:
  - Chinese (PRC): gb2312, gbk, gb18030, big5hkscs, hz
  - Chinese (ROC): big5, cp950
Some other new encodings were added: HP Roman8, ISO_8859-11, ISO_8859-16, PCTP-154, and TIS-620.

The UTF-8 and UTF-16 codecs now cope better with receiving partial input. Previously the StreamReader class would try to read more data, making it impossible to resume decoding from the stream. The read() method will now return as much data as it can and future calls will resume decoding where previous ones left off. (Implemented by Walter Dörwald.)

There is a new collections module for various specialized collection datatypes. Currently it contains just one type, deque, a double-ended queue that supports efficiently adding and removing elements from either end:

```python
>>> from collections import deque
>>> d = deque('ghi') # make a new deque with three items
>>> d.append('j') # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
>>> d # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop() # return and remove the rightmost item
'j'
>>> d.popleft() # return and remove the leftmost item
'f'
>>> list(d) # list the contents of the deque
['g', 'h', 'i']
>>> 'h' in d # search the deque
True
```

Several modules, such as the Queue and threading modules, now take advantage of collections.deque for improved performance. (Contributed by Raymond Hettinger.)
The ConfigParser classes have been enhanced slightly. The read() method now returns a list of the files that were successfully parsed, and the set() method raises TypeError if passed a value argument that isn’t a string. (Contributed by John Belmonte and David Goodger.)

The curses module now supports the ncurses extension use_default_colors(). On platforms where the terminal supports transparency, this makes it possible to use a transparent background. (Contributed by Jörg Lehmann.)

The difflib module now includes an HtmlDiff class that creates an HTML table showing a side by side comparison of two versions of a text. (Contributed by Dan Gass.)

The email package was updated to version 3.0, which dropped various deprecated APIs and removes support for Python versions earlier than 2.3. The 3.0 version of the package uses a new incremental parser for MIME messages, available in the email.FeedParser module. The new parser doesn’t require reading the entire message into memory, and doesn’t raise exceptions if a message is malformed; instead it records any problems in the defect attribute of the message. (Developed by Anthony Baxter, Barry Warsaw, Thomas Wouters, and others.)

The heapq module has been converted to C. The resulting tenfold improvement in speed makes the module suitable for handling high volumes of data. In addition, the module has two new functions nlargest() and nsmallest() that use heaps to find the N largest or smallest values in a dataset without the expense of a full sort. (Contributed by Raymond Hettinger.)
The `httplib` module now contains constants for HTTP status codes defined in various HTTP-related RFC documents. Constants have names such as OK, CREATED, CONTINUE, and MOVED_PERMANENTLY; use pydoc to get a full list. (Contributed by Andrew Eland.)

The `imaplib` module now supports IMAP's THREAD command (contributed by Yves Dionne) and new deleteacl() and myrights() methods (contributed by Arnaud Mazin).

The `itertools` module gained a `groupby(iterable[, *func*])` function. `iterable` is something that can be iterated over to return a stream of elements, and the optional `func` parameter is a function that takes an element and returns a key value; if omitted, the key is simply the element itself. `groupby()` then groups the elements into subsequences which have matching values of the key, and returns a series of 2-tuples containing the key value and an iterator over the subsequence.

Here's an example to make this clearer. The `key` function simply returns whether a number is even or odd, so the result of `groupby()` is to return consecutive runs of odd or even numbers.

```python
>>> import itertools
>>> L = [2, 4, 6, 7, 8, 9, 11, 12, 14]
>>> for key_val, it in itertools.groupby(L, lambda x: x % 2)
...     print key_val, list(it)
... 0 [2, 4, 6]
1 [7]
0 [8]
1 [9, 11]
```
groupby() is typically used with sorted input. The logic for groupby() is similar to the Unix uniq filter which makes it handy for eliminating, counting, or identifying duplicate elements:

```python
>>> word = 'abracadabra'
>>> letters = sorted(word)  # Turn string into a list of letters
['a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r']
>>> for k, g in itertools.groupby(letters):
...     print(k, list(g))
...
'a' ['a', 'a', 'a', 'a', 'a']
'b' ['b', 'b']
'c' ['c']
'd' ['d']
'r' ['r', 'r']
>>> # List unique letters
>>> [k for k, g in groupby(letters)]
['a', 'b', 'c', 'd', 'r']
>>> # Count letter occurrences
>>> [(k, len(list(g))) for k, g in groupby(letters)]
[('a', 5), ('b', 2), ('c', 1), ('d', 1), ('r', 2)]
```

(Contributed by Hye-Shik Chang.)

- `itertools` also gained a function named `tee(iterator, N)` that returns $N$ independent iterators that replicate `iterator`. If $N$ is omitted, the default is 2.

```python
>>> L = [1,2,3]
>>> i1, i2 = itertools.tee(L)
```
Note that `tee()` has to keep copies of the values returned by the iterator; in the worst case, it may need to keep all of them. This should therefore be used carefully if the leading iterator can run far ahead of the trailing iterator in a long stream of inputs. If the separation is large, then you might as well use `list()` instead. When the iterators track closely with one another, `tee()` is ideal. Possible applications include bookmarking, windowing, or lookahead iterators. (Contributed by Raymond Hettinger.)

- A number of functions were added to the `locale` module, such as `bind_textdomain_codeset()` to specify a particular encoding and a family of `gettext()` functions that return messages in the chosen encoding. (Contributed by Gustavo Niemeyer.)

- Some keyword arguments were added to the `logging` package's `basicConfig()` function to simplify log configuration. The default behavior is to log messages to standard error, but various keyword arguments can be specified to log to a particular file, change the logging format, or set the logging level. For example:
Other additions to the `logging` package include a `log(level, msg)` convenience method, as well as a `TimedRotatingFileHandler` class that rotates its log files at a timed interval. The module already had `RotatingFileHandler`, which rotated logs once the file exceeded a certain size. Both classes derive from a new `BaseRotatingHandler` class that can be used to implement other rotating handlers.

(Changes implemented by Vinay Sajip.)

- The `marshal` module now shares interned strings on unpacking a data structure. This may shrink the size of certain pickle strings, but the primary effect is to make `.pyc` files significantly smaller. (Contributed by Martin von Löwis.)

- The `nntplib` module's `NNTP` class gained `description()` and `descriptions()` methods to retrieve newsgroup descriptions for a single group or for a range of groups. (Contributed by Jürgen A. Erhard.)

- Two new functions were added to the `operator` module, `attrgetter(attr)` and `itemgetter(index)`. Both functions return callables that take a single argument and return the corresponding attribute or item; these callables make excellent data extractors when used with `map()` or `sorted()`. For example:

```python
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3)]
>>> map(operator.itemgetter(0), L)
['c', 'd', 'a', 'b']
>>> map(operator.itemgetter(1), L)
[2, 1, 4, 3]
```
The `optparse` module was updated in various ways. The module now passes its messages through `gettext.gettext()`, making it possible to internationalize Optik's help and error messages. Help messages for options can now include the string `'%default'`, which will be replaced by the option's default value. (Contributed by Greg Ward.)

The long-term plan is to deprecate the `rfc822` module in some future Python release in favor of the `email` package. To this end, the `email.Utils.formatdate()` function has been changed to make it usable as a replacement for `rfc822.formatdate()`. You may want to write new e-mail processing code with this in mind. (Change implemented by Anthony Baxter.)

A new `urandom(n)` function was added to the `os` module, returning a string containing `n` bytes of random data. This function provides access to platform-specific sources of randomness such as `/dev/urandom` on Linux or the Windows CryptoAPI. (Contributed by Trevor Perrin.)

Another new function: `os.path.lexists(path)` returns true if the file specified by `path` exists, whether or not it's a symbolic link. This differs from the existing `os.path.exists(path)` function, which returns false if `path` is a symlink that points to a destination that doesn't exist. (Contributed by Beni Cherniavsky.)

A new `getsid()` function was added to the `posix` module that
underlies the os module. (Contributed by J. Raynor.)

- The poplib module now supports POP over SSL. (Contributed by Hector Urtubia.)

- The profile module can now profile C extension functions. (Contributed by Nick Bastin.)

- The random module has a new method called getrandbits(N) that returns a long integer N bits in length. The existing randrange() method now uses getrandbits() where appropriate, making generation of arbitrarily large random numbers more efficient. (Contributed by Raymond Hettinger.)

- The regular expression language accepted by the re module was extended with simple conditional expressions, written as (? (group)A|B). group is either a numeric group ID or a group name defined with (?P<group>...) earlier in the expression. If the specified group matched, the regular expression pattern A will be tested against the string; if the group didn’t match, the pattern B will be used instead. (Contributed by Gustavo Niemeyer.)

- The re module is also no longer recursive, thanks to a massive amount of work by Gustavo Niemeyer. In a recursive regular expression engine, certain patterns result in a large amount of C stack space being consumed, and it was possible to overflow the stack. For example, if you matched a 30000-byte string of a characters against the expression (a|b)+, one stack frame was consumed per character. Python 2.3 tried to check for stack overflow and raise a RuntimeError exception, but certain patterns could sidestep the checking and if you were unlucky Python could segfault. Python 2.4’s regular expression engine can
match this pattern without problems.

- The `signal` module now performs tighter error-checking on the parameters to the `signal.signal()` function. For example, you can’t set a handler on the SIGKILL signal; previous versions of Python would quietly accept this, but 2.4 will raise a `RuntimeError` exception.

- Two new functions were added to the `socket` module. `socketpair()` returns a pair of connected sockets and `getservbyport(port)` looks up the service name for a given port number. (Contributed by Dave Cole and Barry Warsaw.)

- The `sys.exitfunc()` function has been deprecated. Code should be using the existing `atexit` module, which correctly handles calling multiple exit functions. Eventually `sys.exitfunc()` will become a purely internal interface, accessed only by `atexit`.

- The `tarfile` module now generates GNU-format tar files by default. (Contributed by Lars Gustäbel.)

- The `threading` module now has an elegantly simple way to support thread-local data. The module contains a `local` class whose attribute values are local to different threads.

```python
import threading

data = threading.local()
data.number = 42
data.url = ('www.python.org', 80)
```

Other threads can assign and retrieve their own values for the
number and url attributes. You can subclass local to initialize attributes or to add methods. (Contributed by Jim Fulton.)

- The timeit module now automatically disables periodic garbage collection during the timing loop. This change makes consecutive timings more comparable. (Contributed by Raymond Hettinger.)

- The weakref module now supports a wider variety of objects including Python functions, class instances, sets, frozensets, deques, arrays, files, sockets, and regular expression pattern objects. (Contributed by Raymond Hettinger.)

- The xmlrpclib module now supports a multi-call extension for transmitting multiple XML-RPC calls in a single HTTP operation. (Contributed by Brian Quinlan.)

- The mpz, rotor, and xreadlines modules have been removed.

cookielib

The cookielib library supports client-side handling for HTTP cookies, mirroring the Cookie module's server-side cookie support. Cookies are stored in cookie jars; the library transparently stores cookies offered by the web server in the cookie jar, and fetches the cookie from the jar when connecting to the server. As in web browsers, policy objects control whether cookies are accepted or not.

In order to store cookies across sessions, two implementations of cookie jars are provided: one that stores cookies in the Netscape format so applications can use the Mozilla or Lynx cookie files, and one that stores cookies in the same format as the Perl libwww library.
urllib2 has been changed to interact with cookielib: HTTPCookieProcessor manages a cookie jar that is used when accessing URLs.

This module was contributed by John J. Lee.

doctest

The doctest module underwent considerable refactoring thanks to Edward Loper and Tim Peters. Testing can still be as simple as running doctest.testmod(), but the refactorings allow customizing the module’s operation in various ways.

The new DocTestFinder class extracts the tests from a given object’s docstrings:

```python
def f(x, y):
    """""">>> f(2,2)
    4
    >>> f(3,2)
    6
    """
    return x*y

finder = doctest.DocTestFinder()

# Get list of DocTest instances
tests = finder.find(f)
```

The new DocTestRunner class then runs individual tests and can produce a summary of the results:

```python
runner = doctest.DocTestRunner()
for t in tests:
```
tried, failed = runner.run(t)
runner.summarize(verbos=1)

The above example produces the following output:

```
1 items passed all tests:
   2 tests in f
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

DocTestRunner uses an instance of the OutputChecker class to compare the expected output with the actual output. This class takes a number of different flags that customize its behaviour; ambitious users can also write a completely new subclass of OutputChecker.

The default output checker provides a number of handy features. For example, with the `doctest.ELLIPSIS` option flag, an ellipsis (...) in the expected output matches any substring, making it easier to accommodate outputs that vary in minor ways:

```
def o(n):
    """>>> o(1)
<__main__.C instance at 0x...>
>>> """
```

Another special string, `<BLANKLINE>`, matches a blank line:

```
def p(n):
    """>>> p(1)
<BLANKLINE>
>>> """
Another new capability is producing a diff-style display of the output by specifying the `doctest.REPORT_UDIFF` (unified diffs), `doctest.REPORT_CDIFF` (context diffs), or `doctest.REPORT_NDIFF` (delta-style) option flags. For example:

```python
def g(n):
    """>> g(4)
here
is
a
lengthy

L = 'here is a rather lengthy list of words'.split
for word in L[:n]:
    print word
```

Running the above function’s tests with `doctest.REPORT_UDIFF` specified, you get the following output:

```
**********************************************************************
File "t.py", line 15, in g
Failed example:
g(4)
Differences (unified diff with -expected +actual):
@@ -2,3 +2,3 @@
is
+a
-lengthy
+rather
**********************************************************************
```
Build and C API Changes

Some of the changes to Python’s build process and to the C API are:

- Three new convenience macros were added for common return values from extension functions: `Py_RETURN_NONE`, `Py_RETURN_TRUE`, and `Py_RETURN_FALSE`. (Contributed by Brett Cannon.)
- Another new macro, `Py_CLEAR(obj)`, decreases the reference count of `obj` and sets `obj` to the null pointer. (Contributed by Jim Fulton.)
- A new function, `PyTuple_Pack(N, obj1, obj2, ..., objN)`, constructs tuples from a variable length argument list of Python objects. (Contributed by Raymond Hettinger.)
- A new function, `PyDict_Contains(d, k)`, implements fast dictionary lookups without masking exceptions raised during the look-up process. (Contributed by Raymond Hettinger.)
- The `Py_IS_NAN(X)` macro returns 1 if its float or double argument `X` is a NaN. (Contributed by Tim Peters.)
- C code can avoid unnecessary locking by using the new `PyEval_ThreadsInitialized()` function to tell if any thread operations have been performed. If this function returns false, no lock operations are needed. (Contributed by Nick Coghlan.)
- A new function, `PyArg_VaParseTupleAndKeywords()`, is the same as `PyArg_ParseTupleAndKeywords()` but takes a `va_list` instead of a number of arguments. (Contributed by Greg Chapman.)
- A new method flag, `METH_COEXISTS`, allows a function defined in slots to co-exist with a `PyCFunction` having the same name. This can halve the access time for a method such as `set.__contains__()`.(Contributed by Raymond Hettinger.)
- Python can now be built with additional profiling for the interpreter.
itself, intended as an aid to people developing the Python core. Providing --enable-profiling to the configure script will let you profile the interpreter with gprof, and providing the --with-tsc switch enables profiling using the Pentium’s Time-Stamp-Counter register. Note that the --with-tsc switch is slightly misnamed, because the profiling feature also works on the PowerPC platform, though that processor architecture doesn’t call that register “the TSC register”. (Contributed by Jeremy Hylton.)

- The tracebackobject type has been renamed to PyTracebackObject.

Port-Specific Changes

- The Windows port now builds under MSVC++ 7.1 as well as version 6. (Contributed by Martin von Löwis.)
Porting to Python 2.4

This section lists previously described changes that may require changes to your code:

- Left shifts and hexadecimal/octet constants that are too large no longer trigger a `FutureWarning` and return a value limited to 32 or 64 bits; instead they return a long integer.
- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.
- The `zip()` built-in function and `itertools.izip()` now return an empty list instead of raising a `TypeError` exception if called with no arguments.
- You can no longer compare the date and `datetime` instances provided by the `datetime` module. Two instances of different classes will now always be unequal, and relative comparisons (<, >) will raise a `TypeError`.
- `dircache.listdir()` now passes exceptions to the caller instead of returning empty lists.
- `LexicalHandler.startDTD()` used to receive the public and system IDs in the wrong order. This has been corrected; applications relying on the wrong order need to be fixed.
- `fcntl.ioctl()` now warns if the `mutate` argument is omitted and relevant.
- The `tarfile` module now generates GNU-format tar files by default.
- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`.
- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error.
- The `signals.signal()` function now raises a `RuntimeError` exception for certain illegal values; previously these errors would
pass silently. For example, you can no longer set a handler on the SIGKILL signal.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Koray Can, Hye-Shik Chang, Michael Dyck, Raymond Hettinger, Brian Hurt, Hamish Lawson, Fredrik Lundh, Sean Reifschneider, Sadruddin Rejeb.
This article explains the new features in Python 2.3. Python 2.3 was released on July 29, 2003.

The main themes for Python 2.3 are polishing some of the features added in 2.2, adding various small but useful enhancements to the core language, and expanding the standard library. The new object model introduced in the previous version has benefited from 18 months of bugfixes and from optimization efforts that have improved the performance of new-style classes. A few new built-in functions have been added such as `sum()` and `enumerate()`. The `in` operator can now be used for substring searches (e.g.  "ab" in "abc" returns `True`).

Some of the many new library features include Boolean, set, heap, and date/time data types, the ability to import modules from ZIP-format archives, metadata support for the long-awaited Python catalog, an updated version of IDLE, and modules for logging messages, wrapping text, parsing CSV files, processing command-line options, using BerkeleyDB databases... the list of new and enhanced modules is lengthy.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.3, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.
PEP 218: A Standard Set Datatype

The new sets module contains an implementation of a set datatype. The Set class is for mutable sets, sets that can have members added and removed. The ImmutableSet class is for sets that can't be modified, and instances of ImmutableSet can therefore be used as dictionary keys. Sets are built on top of dictionaries, so the elements within a set must be hashable.

Here's a simple example:

```python
>>> import sets
>>> S = sets.Set([1, 2, 3])
>>> S
Set([1, 2, 3])
>>> 1 in S
True
>>> 0 in S
False
>>> S.add(5)
>>> S.remove(3)
>>> S
Set([1, 2, 5])
```

The union and intersection of sets can be computed with the union() and intersection() methods; an alternative notation uses the bitwise operators `&` and `|`. Mutable sets also have in-place versions of these methods, union_update() and intersection_update().

```python
>>> S1 = sets.Set([1, 2, 3])
>>> S2 = sets.Set([4, 5, 6])
>>> S1.union(S2)
```
It's also possible to take the symmetric difference of two sets. This is the set of all elements in the union that aren't in the intersection. Another way of putting it is that the symmetric difference contains all elements that are in exactly one set. Again, there's an alternative notation (^), and an in-place version with the ungainly name symmetric_difference_update().

There are also issubset() and issuperset() methods for checking whether one set is a subset or superset of another:
False
>>> S1.issuperset(S2)
True

See also:

**PEP 218 - Adding a Built-In Set Object Type**

PEP written by Greg V. Wilson. Implemented by Greg V. Wilson, Alex Martelli, and GvR.
PEP 255: Simple Generators

In Python 2.2, generators were added as an optional feature, to be enabled by a `from __future__ import generators` directive. In 2.3 generators no longer need to be specially enabled, and are now always present; this means that `yield` is now always a keyword. The rest of this section is a copy of the description of generators from the “What’s New in Python 2.2” document; if you read it back when Python 2.2 came out, you can skip the rest of this section.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a `yield` statement is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn’t return a single value;
instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator’s state of execution is suspended and local variables are preserved. On the next call to the generator’s `.next()` method, the function will resume executing immediately after the `yield` statement. (For complicated reasons, the `yield` statement isn’t allowed inside the `try` block of a `try...finally` statement; read PEP 255 for a full explanation of the interaction between `yield` and exceptions.)

Here’s a sample usage of the `generate_ints()` generator:

```python
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write for `i` in `generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values;
afterwards the generator cannot return any further values. return with a value, such as return 5, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising StopIteration manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting self.count to 0, and having the next() method increment self.count and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. Lib/test/test_generators.py contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in Lib/test/test_generators.py produce solutions for the N-Queens problem (placing $N$ queens on an $N\times N$ chess board so that no queen threatens another) and the Knight's Tour (a route that takes a knight to every square of an $N\times N$ chessboard without visiting any square twice).

The idea of generators comes from other programming languages,
especially Icon (https://www.cs.arizona.edu/icon/), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon Programming Language” at https://www.cs.arizona.edu/icon/docs/ipd266.htm gives an idea of what this looks like:

```plaintext
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

In Icon the find() function returns the indexes at which the substring “or” is found: 3, 23, 33. In the if statement, i is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered part of the core Python language, but learning or using them isn’t compulsory; if they don’t solve any problems that you have, feel free to ignore them. One novel feature of Python’s interface as compared to Icon’s is that a generator’s state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See also:

**PEP 255 - Simple Generators**

Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.
PEP 263: Source Code Encodings

Python source files can now be declared as being in different character set encodings. Encodings are declared by including a specially formatted comment in the first or second line of the source file. For example, a UTF-8 file can be declared with:

```bash
#!/usr/bin/env python
#	-*- coding: UTF-8 -*-
```

Without such an encoding declaration, the default encoding used is 7-bit ASCII. Executing or importing modules that contain string literals with 8-bit characters and have no encoding declaration will result in a `DeprecationWarning` being signalled by Python 2.3; in 2.4 this will be a syntax error.

The encoding declaration only affects Unicode string literals, which will be converted to Unicode using the specified encoding. Note that Python identifiers are still restricted to ASCII characters, so you can’t have variable names that use characters outside of the usual alphanumerics.

See also:

**PEP 263 - Defining Python Source Code Encodings**
Written by Marc-André Lemburg and Martin von Löwis; implemented by Suzuki Hisao and Martin von Löwis.
PEP 273: Importing Modules from ZIP Archives

The new `zipimport` module adds support for importing modules from a ZIP-format archive. You don't need to import the module explicitly; it will be automatically imported if a ZIP archive's filename is added to `sys.path`. For example:

```
unzip -l /tmp/example.zip
Archive: /tmp/example.zip
   Length     Date       Time    Name
     ---- ------------------ ----
  8467 11-26-02 22:30 jwzthreading.py
     ---- -------------------
  8467               1 file

./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip') # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
``` 

An entry in `sys.path` can now be the filename of a ZIP archive. The ZIP archive can contain any kind of files, but only files named `*.py`, `*.pyc`, or `*.pyo` can be imported. If an archive only contains `*.py` files, Python will not attempt to modify the archive by adding the corresponding `*.pyc` file, meaning that if a ZIP archive doesn't contain `*.pyc` files, importing may be rather slow.

A path within the archive can also be specified to only import from a
subdirectory; for example, the path `/tmp/example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

See also:

**PEP 273 - Import Modules from Zip Archives**
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302. See section PEP 302: New Import Hooks for a description of the new import hooks.
PEP 277: Unicode file name support for Windows NT

On Windows NT, 2000, and XP, the system stores file names as Unicode strings. Traditionally, Python has represented file names as byte strings, which is inadequate because it renders some file names inaccessible.

Python now allows using arbitrary Unicode strings (within the limitations of the file system) for all functions that expect file names, most notably the `open()` built-in function. If a Unicode string is passed to `os.listdir()`, Python now returns a list of Unicode strings. A new function, `os.getcwd()`, returns the current directory as a Unicode string.

Byte strings still work as file names, and on Windows Python will transparently convert them to Unicode using the `mbcs` encoding.

Other systems also allow Unicode strings as file names but convert them to byte strings before passing them to the system, which can cause a `UnicodeError` to be raised. Applications can test whether arbitrary Unicode strings are supported as file names by checking `os.path.supports_unicode_filenames`, a Boolean value.

Under MacOS, `os.listdir()` may now return Unicode filenames.

See also:

**PEP 277 - Unicode file name support for Windows NT**
Written by Neil Hodgson; implemented by Neil Hodgson, Martin von Löwis, and Mark Hammond.
PEP 278: Universal Newline Support

The three major operating systems used today are Microsoft Windows, Apple's Macintosh OS, and the various Unix derivatives. A minor irritation of cross-platform work is that these three platforms all use different characters to mark the ends of lines in text files. Unix uses the linefeed (ASCII character 10), MacOS uses the carriage return (ASCII character 13), and Windows uses a two-character sequence of a carriage return plus a newline.

Python's file objects can now support end of line conventions other than the one followed by the platform on which Python is running. Opening a file with the mode `''U''` or `''rU''` will open a file for reading in universal newlines mode. All three line ending conventions will be translated to a `'\n'` in the strings returned by the various file methods such as `read()` and `readline()`.

Universal newline support is also used when importing modules and when executing a file with the `execfile()` function. This means that Python modules can be shared between all three operating systems without needing to convert the line-endings.

This feature can be disabled when compiling Python by specifying the `--without-universal-newlines` switch when running Python's `configure` script.

See also:

**PEP 278 - Universal Newline Support**
Written and implemented by Jack Jansen.
PEP 279: enumerate()

A new built-in function, `enumerate()`, will make certain loops a bit clearer. `enumerate(thing)`, where `thing` is either an iterator or a sequence, returns an iterator that will return `(0, thing[0]), (1, thing[1]), (2, thing[2]),` and so forth.

A common idiom to change every element of a list looks like this:

```python
for i in range(len(L)):
    item = L[i]
    # ... compute some result based on item ...
    L[i] = result
```

This can be rewritten using `enumerate()` as:

```python
for i, item in enumerate(L):
    # ... compute some result based on item ...
    L[i] = result
```

See also:

**PEP 279 - The enumerate() built-in function**

Written and implemented by Raymond D. Hettinger.
PEP 282: The logging Package

A standard package for writing logs, `logging`, has been added to Python 2.3. It provides a powerful and flexible mechanism for generating logging output which can then be filtered and processed in various ways. A configuration file written in a standard format can be used to control the logging behavior of a program. Python includes handlers that will write log records to standard error or to a file or socket, send them to the system log, or even e-mail them to a particular address; of course, it's also possible to write your own handler classes.

The Logger class is the primary class. Most application code will deal with one or more Logger objects, each one used by a particular subsystem of the application. Each Logger is identified by a name, and names are organized into a hierarchy using `.` as the component separator. For example, you might have Logger instances named `server`, `server.auth` and `server.network`. The latter two instances are below `server` in the hierarchy. This means that if you turn up the verbosity for `server` or direct `server` messages to a different handler, the changes will also apply to records logged to `server.auth` and `server.network`. There's also a root Logger that's the parent of all other loggers.

For simple uses, the `logging` package contains some convenience functions that always use the root log:

```python
import logging

logging.debug('Debugging information')
logging.info('Informational message')
```
logging.warning('Warning: config file %s not found',)
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')

This produces the following output:

WARNING: root: Warning: config file server.conf not found
ERROR: root: Error occurred
CRITICAL: root: Critical error -- shutting down

In the default configuration, informational and debugging messages are suppressed and the output is sent to standard error. You can enable the display of informational and debugging messages by calling the setLevel() method on the root logger.

Notice the warning() call's use of string formatting operators; all of the functions for logging messages take the arguments \( \text{msg, arg1, arg2, ...} \) and log the string resulting from \( \text{msg} \% \text{(arg1, arg2, ...)} \).

There's also an exception() function that records the most recent traceback. Any of the other functions will also record the traceback if you specify a true value for the keyword argument \text{exc_info}.

```python
def f():
    try:
        1/0
    except:
        logging.exception('Problem recorded')
f()
```

This produces the following output:

ERROR: root: Problem recorded
Slightly more advanced programs will use a logger other than the root logger. The `getLogger(name)` function is used to get a particular log, creating it if it doesn’t exist yet. `getLogger(None)` returns the root logger.

```python
log = logging.getLogger('server')
...
log.info('Listening on port %i', port)
...
log.critical('Disk full')
...```

Log records are usually propagated up the hierarchy, so a message logged to `server.auth` is also seen by `server` and `root`, but a Logger can prevent this by setting its `propagate` attribute to `False`.

There are more classes provided by the `logging` package that can be customized. When a Logger instance is told to log a message, it creates a LogRecord instance that is sent to any number of different Handler instances. Loggers and handlers can also have an attached list of filters, and each filter can cause the LogRecord to be ignored or can modify the record before passing it along. When they’re finally output, LogRecord instances are converted to text by a Formatter class. All of these classes can be replaced by your own specially-written classes.

With all of these features the `logging` package should provide enough flexibility for even the most complicated applications. This is
only an incomplete overview of its features, so please see the package’s reference documentation for all of the details. Reading PEP 282 will also be helpful.

**See also:**

**PEP 282 - A Logging System**  
Written by Vinay Sajip and Trent Mick; implemented by Vinay Sajip.
PEP 285: A Boolean Type

A Boolean type was added to Python 2.3. Two new constants were added to the __builtin__ module, True and False. (True and False constants were added to the built-ins in Python 2.2.1, but the 2.2.1 versions are simply set to integer values of 1 and 0 and aren’t a different type.)

The type object for this new type is named bool; the constructor for it takes any Python value and converts it to True or False.

```python
globals() 'undefined variable
>>> bool(1)
True
>>> bool(0)
False
>>> bool([])
False
>>> bool((1,))
True
```

Most of the standard library modules and built-in functions have been changed to return Booleans.

```python
>>> obj = []
>>> hasattr(obj, 'append')
True
>>> isinstance(obj, list)
True
>>> isinstance(obj, tuple)
False
```

Python’s Booleans were added with the primary goal of making code clearer. For example, if you’re reading a function and encounter the
statement `return 1`, you might wonder whether the `1` represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is `return True`, however, the meaning of the return value is quite clear.

Python's Booleans were not added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an `if` statement always evaluate to a Boolean result. Python is not this strict and never will be, as PEP 285 explicitly says. This means you can still use any expression in an `if` statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the `int` class so that arithmetic using a Boolean still works.

```python
>>> True + 1
2
>>> False + 1
1
>>> False * 75
0
>>> True * 75
75
```

To sum up `True` and `False` in a sentence: they're alternative ways to spell the integer values 1 and 0, with the single difference that `str()` and `repr()` return the strings 'True' and 'False' instead of '1' and '0'.

**See also:**

**PEP 285 - Adding a bool type**

Written and implemented by GvR.
PEP 293: Codec Error Handling Callbacks

When encoding a Unicode string into a byte string, unencodable characters may be encountered. So far, Python has allowed specifying the error processing as either “strict” (raising `UnicodeError`), “ignore” (skipping the character), or “replace” (using a question mark in the output string), with “strict” being the default behavior. It may be desirable to specify alternative processing of such errors, such as inserting an XML character reference or HTML entity reference into the converted string.

Python now has a flexible framework to add different processing strategies. New error handlers can be added with `codecs.register_error()`, and codecs then can access the error handler with `codecs.lookup_error()`. An equivalent C API has been added for codecs written in C. The error handler gets the necessary state information such as the string being converted, the position in the string where the error was detected, and the target encoding. The handler can then either raise an exception or return a replacement string.

Two additional error handlers have been implemented using this framework: “backslashreplace” uses Python backslash quoting to represent unencodable characters and “xmlcharrefreplace” emits XML character references.

See also:

**PEP 293 - Codec Error Handling Callbacks**

Written and implemented by Walter Dörwald.
PEP 301: Package Index and Metadata for Distutils

Support for the long-requested Python catalog makes its first appearance in 2.3.

The heart of the catalog is the new Distutils register command. Running `python setup.py register` will collect the metadata describing a package, such as its name, version, maintainer, description, &c., and send it to a central catalog server. The resulting catalog is available from https://pypi.org.

To make the catalog a bit more useful, a new optional classifiers keyword argument has been added to the Distutils setup() function. A list of Trove-style strings can be supplied to help classify the software.

Here's an example setup.py with classifiers, written to be compatible with older versions of the Distutils:

```python
from distutils import core
kw = {
    'name': 'Quixote',
    'version': '0.5.1',
    'description': 'A highly Pythonic Web applicat
    # ...
}

if (hasattr(core, 'setup_keywords') and
    'classifiers' in core.setup_keywords):
    kw['classifiers'] = \
        ['Topic :: Internet :: WWW/HTTP :: Dynamic C
            'Environment :: No Input/Output (Daemon)',
            'Intended Audience :: Developers'],
```
core.setup(**kw)

The full list of classifiers can be obtained by running `python setup.py register --list-classifiers`.

See also:

**PEP 301 - Package Index and Metadata for Distutils**
Written and implemented by Richard Jones.
PEP 302: New Import Hooks

While it's been possible to write custom import hooks ever since the ihooks module was introduced in Python 1.3, no one has ever been really happy with it because writing new import hooks is difficult and messy. There have been various proposed alternatives such as the imputil and iu modules, but none of them has ever gained much acceptance, and none of them were easily usable from C code.

PEP 302 borrows ideas from its predecessors, especially from Gordon McMillan's iu module. Three new items are added to the sys module:

- `sys.path_hooks` is a list of callable objects; most often they'll be classes. Each callable takes a string containing a path and either returns an importer object that will handle imports from this path or raises an `ImportError` exception if it can't handle this path.
- `sys.path_importer_cache` caches importer objects for each path, so `sys.path_hooks` will only need to be traversed once for each path.
- `sys.meta_path` is a list of importer objects that will be traversed before `sys.path` is checked. This list is initially empty, but user code can add objects to it. Additional built-in and frozen modules can be imported by an object added to this list.

Importer objects must have a single method, `find_module(fullname, path=None)`. `fullname` will be a module or package name, e.g. `string` or `distutils.core`. `find_module()` must return a loader object that has a single method, `load_module(fullname)`, that creates and returns the corresponding module object.
Pseudo-code for Python’s new import logic, therefore, looks something like this (simplified a bit; see PEP 302 for the full details):

```python
for mp in sys.meta_path:
    loader = mp(fullname)
    if loader is not None:
        <module> = loader.load_module(fullname)

for path in sys.path:
    for hook in sys.path_hooks:
        try:
            importer = hook(path)
        except ImportError:
            # ImportError, so try the other path hooks
            pass
        else:
            loader = importer.find_module(fullname)
            <module> = loader.load_module(fullname)

# Not found!
raise ImportError
```

See also:

**PEP 302 - New Import Hooks**

Written by Just van Rossum and Paul Moore. Implemented by Just van Rossum.
Comma-separated files are a format frequently used for exporting data from databases and spreadsheets. Python 2.3 adds a parser for comma-separated files.

Comma-separated format is deceptively simple at first glance:

```
Costs,150,200,3.95
```

Read a line and call `line.split(',')`: what could be simpler? But toss in string data that can contain commas, and things get more complicated:

```
"Costs",150,200,3.95,"Includes taxes, shipping, and"
```

A big ugly regular expression can parse this, but using the new `csv` package is much simpler:

```python
import csv

input = open('datafile', 'rb')
reader = csv.reader(input)
for line in reader:
    print line
```

The `reader()` function takes a number of different options. The field separator isn’t limited to the comma and can be changed to any character, and so can the quoting and line-ending characters.

Different dialects of comma-separated files can be defined and registered; currently there are two dialects, both used by Microsoft
Excel. A separate `csv.writer` class will generate comma-separated files from a succession of tuples or lists, quoting strings that contain the delimiter.

**See also:**

**PEP 305 - CSV File API**

Written and implemented by Kevin Altis, Dave Cole, Andrew McNamara, Skip Montanaro, Cliff Wells.
PEP 307: Pickle Enhancements

The pickle and cPickle modules received some attention during the 2.3 development cycle. In 2.2, new-style classes could be pickled without difficulty, but they weren’t pickled very compactly; PEP 307 quotes a trivial example where a new-style class results in a pickled string three times longer than that for a classic class.

The solution was to invent a new pickle protocol. The pickle.dumps() function has supported a text-or-binary flag for a long time. In 2.3, this flag is redefined from a Boolean to an integer: 0 is the old text-mode pickle format, 1 is the old binary format, and now 2 is a new 2.3-specific format. A new constant, pickle.HIGHEST_PROTOCOL, can be used to select the fanciest protocol available.

Unpickling is no longer considered a safe operation. 2.2’s pickle provided hooks for trying to prevent unsafe classes from being unpickled (specifically, a __safe_for_unpickling__ attribute), but none of this code was ever audited and therefore it’s all been ripped out in 2.3. You should not unpickle untrusted data in any version of Python.

To reduce the pickling overhead for new-style classes, a new interface for customizing pickling was added using three special methods: __getstate__(), __setstate__(), and __getnewargs__(). Consult PEP 307 for the full semantics of these methods.

As a way to compress pickles yet further, it’s now possible to use integer codes instead of long strings to identify pickled classes. The Python Software Foundation will maintain a list of standardized codes; there’s also a range of codes for private use. Currently no codes have
been specified.

See also:

**PEP 307 - Extensions to the pickle protocol**

Written and implemented by Guido van Rossum and Tim Peters.
Extended Slices

Ever since Python 1.4, the slicing syntax has supported an optional third “step” or “stride” argument. For example, these are all legal Python syntax: L[1:10:2], L[:-1:1], L[::-1]. This was added to Python at the request of the developers of Numerical Python, which uses the third argument extensively. However, Python’s built-in list, tuple, and string sequence types have never supported this feature, raising a TypeError if you tried it. Michael Hudson contributed a patch to fix this shortcoming.

For example, you can now easily extract the elements of a list that have even indexes:

```python
>>> L = range(10)
>>> L[::2]
[0, 2, 4, 6, 8]
```

Negative values also work to make a copy of the same list in reverse order:

```python
>>> L[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

This also works for tuples, arrays, and strings:

```python
>>> s='abcd'
>>> s[::2]
'ac'
>>> s[::-1]
'dcba'
```

If you have a mutable sequence such as a list or an array you can
assign to or delete an extended slice, but there are some differences between assignment to extended and regular slices. Assignment to a regular slice can be used to change the length of the sequence:

```python
>>> a = range(3)
>>> a
[0, 1, 2]
>>> a[1:3] = [4, 5, 6]
>>> a
[0, 4, 5, 6]
```

Extended slices aren't this flexible. When assigning to an extended slice, the list on the right hand side of the statement must contain the same number of items as the slice it is replacing:

```python
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> a[::2] = [0, -1]
>>> a
[0, 1, -1, 3]
>>> a[::2] = [0, 1, 2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
```

Deletion is more straightforward:

```python
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
```
>>> del a[::2]

One can also now pass slice objects to the \texttt{\_\_getitem\_()} methods of the built-in sequences:

```python
>>> range(10).\_\_getitem\_\((\texttt{slice(0, 5, 2)})

Or use slice objects directly in subscripts:

```python
>>> range(10)[\texttt{slice(0, 5, 2)}]
```

To simplify implementing sequences that support extended slicing, slice objects now have a method \texttt{indices(length)} which, given the length of a sequence, returns a \texttt{(start, stop, step)} tuple that can be passed directly to \texttt{range().indices()} handles omitted and out-of-bounds indices in a manner consistent with regular slices (and this innocuous phrase hides a welter of confusing details!). The method is intended to be used like this:

```python
class FakeSeq:
    ...
    def calc_item(self, i):
        ...
        def __getitem__(self, item):
            if isinstance(item, slice):
                indices = item.indices(len(self))
                return FakeSeq([self.calc_item(i) for i
            else:
                return self.calc_item(i)
```

```
From this example you can also see that the built-in \texttt{slice} object is now the type object for the slice type, and is no longer a function. This is consistent with Python 2.2, where \texttt{int}, \texttt{str}, etc., underwent the same change.
Other Language Changes

Here are all of the changes that Python 2.3 makes to the core Python language.

- The `yield` statement is now always a keyword, as described in section PEP 255: Simple Generators of this document.

- A new built-in function `enumerate()` was added, as described in section PEP 279: enumerate() of this document.

- Two new constants, `True` and `False` were added along with the built-in `bool` type, as described in section PEP 285: A Boolean Type of this document.

- The `int()` type constructor will now return a long integer instead of raising an `OverflowError` when a string or floating-point number is too large to fit into an integer. This can lead to the paradoxical result that `isinstance(int(expression), int)` is false, but that seems unlikely to cause problems in practice.

- Built-in types now support the extended slicing syntax, as described in section Extended Slices of this document.

- A new built-in function, `sum(iterable, start=0)`, adds up the numeric items in the iterable object and returns their sum. `sum()` only accepts numbers, meaning that you can't use it to concatenate a bunch of strings. (Contributed by Alex Martelli.)

- `list.insert(pos, value)` used to insert `value` at the front of the list when `pos` was negative. The behaviour has now been
changed to be consistent with slice indexing, so when pos is -1 the value will be inserted before the last element, and so forth.

- `list.index(value)`, which searches for `value` within the list and returns its index, now takes optional `start` and `stop` arguments to limit the search to only part of the list.

- Dictionaries have a new method, `pop(key[, *default*])`, that returns the value corresponding to `key` and removes that key/value pair from the dictionary. If the requested key isn’t present in the dictionary, `default` is returned if it’s specified and `KeyError` raised if it isn’t.

```python
>>> d = {1:2}
>>> d
{1: 2}
>>> d.pop(4)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 4
>>> d.pop(1)
2
>>> d.pop(1)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 'pop(): dictionary is empty'
>>> d
{}
```  

There’s also a new class method, `dict.fromkeys(iterable, value)`, that creates a dictionary with keys taken from the supplied iterator `iterable` and all values set to `value`, defaulting to `None`.  

Also, the `dict()` constructor now accepts keyword arguments to simplify creating small dictionaries:

```python
>>> dict(red=1, blue=2, green=3, black=4)
{'blue': 2, 'black': 4, 'green': 3, 'red': 1}
```

(Contributed by Just van Rossum.)

- The `assert` statement no longer checks the `__debug__` flag, so you can no longer disable assertions by assigning to `__debug__`. Running Python with the `-O` switch will still generate code that doesn’t execute any assertions.

- Most type objects are now callable, so you can use them to create new objects such as functions, classes, and modules. (This means that the new module can be deprecated in a future Python version, because you can now use the type objects available in the `types` module.) For example, you can create a new module object with the following code:

```python
>>> import types
>>> m = types.ModuleType('abc', 'docstring')
>>> m
<module 'abc' (built-in)>
>>> m.__doc__
'docstring'
```

- A new warning, `PendingDeprecationWarning` was added to indicate features which are in the process of being deprecated. The warning will not be printed by default. To check for use of features that will be deprecated in the future, supply -
always::PendingDeprecationWarning:: on the command line or use warnings.filterwarnings().

- The process of deprecating string-based exceptions, as in `raise "Error occurred"`, has begun. Raising a string will now trigger `PendingDeprecationWarning`.

- Using `None` as a variable name will now result in a `SyntaxWarning` warning. In a future version of Python, `None` may finally become a keyword.

- The `xreadlines()` method of file objects, introduced in Python 2.1, is no longer necessary because files now behave as their own iterator. `xreadlines()` was originally introduced as a faster way to loop over all the lines in a file, but now you can simply write `for line in file_obj`. File objects also have a new read-only encoding attribute that gives the encoding used by the file; Unicode strings written to the file will be automatically converted to bytes using the given encoding.

- The method resolution order used by new-style classes has changed, though you'll only notice the difference if you have a really complicated inheritance hierarchy. Classic classes are unaffected by this change. Python 2.2 originally used a topological sort of a class's ancestors, but 2.3 now uses the C3 algorithm as described in the paper “A Monotonic Superclass Linearization for Dylan”. To understand the motivation for this change, read Michele Simionato's article “Python 2.3 Method Resolution Order”, or read the thread on python-dev starting with the message at https://mail.python.org/pipermail/python-dev/2002-October/029035.html. Samuele Pedroni first pointed out the problem and also implemented the fix by coding the C3 algorithm.
• Python runs multithreaded programs by switching between threads after executing N bytecodes. The default value for N has been increased from 10 to 100 bytecodes, speeding up single-threaded applications by reducing the switching overhead. Some multithreaded applications may suffer slower response time, but that's easily fixed by setting the limit back to a lower number using `sys.setcheckinterval(N)`. The limit can be retrieved with the new `sys.getcheckinterval()` function.

• One minor but far-reaching change is that the names of extension types defined by the modules included with Python now contain the module and a '.' in front of the type name. For example, in Python 2.2, if you created a socket and printed its `__class__`, you'd get this output:

```python
>>> s = socket.socket()
>>> s.__class__
<type 'socket'>
```

In 2.3, you get this:

```python
>>> s.__class__
<type '_socket.socket'>
```

• One of the noted incompatibilities between old- and new-style classes has been removed: you can now assign to the `__name__` and `__bases__` attributes of new-style classes. There are some restrictions on what can be assigned to `__bases__` along the lines of those relating to assigning to an instance's `__class__` attribute.

String Changes
The `in` operator now works differently for strings. Previously, when evaluating `X in Y` where `X` and `Y` are strings, `X` could only be a single character. That’s now changed; `X` can be a string of any length, and `X in Y` will return `True` if `X` is a substring of `Y`. If `X` is the empty string, the result is always `True`.

```
>>> 'ab' in 'abcd'
True
>>> 'ad' in 'abcd'
False
>>> '' in 'abcd'
True
```

Note that this doesn’t tell you where the substring starts; if you need that information, use the `find()` string method.

- The `strip()`, `lstrip()`, and `rstrip()` string methods now have an optional argument for specifying the characters to strip. The default is still to remove all whitespace characters:

```
>>> ' abc '.strip()
'abc'
>>> '<<><abc><><>' .strip('<<')
'abc'
>>> '<<><abc><><>

'.strip('<<')
'abc

'<<><abc><><>

u'\u4000\u4001abc\u4000'.strip(u'\u4000')
u'\u4001abc'

```

(Suggested by Simon Brunning and implemented by Walter Dörwald.)

- The `startswith()` and `endswith()` string methods now
accept negative numbers for the *start* and *end* parameters.

- Another new string method is `zfill()`, originally a function in the `string` module. `zfill()` pads a numeric string with zeros on the left until it's the specified width. Note that the `%` operator is still more flexible and powerful than `zfill()`.

```
>>> '45'.zfill(4)
'0045'
>>> '12345'.zfill(4)
'12345'
>>> 'goofy'.zfill(6)
'0goofy'
```

(Contributed by Walter Dörwald.)

- A new type object, `basestring`, has been added. Both 8-bit strings and Unicode strings inherit from this type, so `isinstance(obj, basestring)` will return `True` for either kind of string. It's a completely abstract type, so you can't create `basestring` instances.

- Interned strings are no longer immortal and will now be garbage-collected in the usual way when the only reference to them is from the internal dictionary of interned strings. (Implemented by Oren Tirosh.)

**Optimizations**

- The creation of new-style class instances has been made much faster; they're now faster than classic classes!
- The `sort()` method of list objects has been extensively rewritten by Tim Peters, and the implementation is significantly faster.
Multiplication of large long integers is now much faster thanks to an implementation of Karatsuba multiplication, an algorithm that scales better than the O(n^2) required for the grade-school multiplication algorithm. (Original patch by Christopher A. Craig, and significantly reworked by Tim Peters.)

- The SET_LINENO opcode is now gone. This may provide a small speed increase, depending on your compiler’s idiosyncrasies. See section Other Changes and Fixes for a longer explanation. (Removed by Michael Hudson.)

- `xrange()` objects now have their own iterator, making `for i in xrange(n)` slightly faster than `for i in range(n)`. (Patch by Raymond Hettinger.)

- A number of small rearrangements have been made in various hotspots to improve performance, such as inlining a function or removing some code. (Implemented mostly by GvR, but lots of people have contributed single changes.)

The net result of the 2.3 optimizations is that Python 2.3 runs the pystone benchmark around 25% faster than Python 2.2.
New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the Misc/NEWS file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The `array` module now supports arrays of Unicode characters using the 'u' format character. Arrays also now support using the `+=` assignment operator to add another array's contents, and the `*=` assignment operator to repeat an array. (Contributed by Jason Orendorff.)

- The `bsddb` module has been replaced by version 4.1.6 of the PyBSDDDB package, providing a more complete interface to the transactional features of the BerkeleyDB library.

The old version of the module has been renamed to `bsddb185` and is no longer built automatically; you'll have to edit `Modules/Setup` to enable it. Note that the new `bsddb` package is intended to be compatible with the old module, so be sure to file bugs if you discover any incompatibilities. When upgrading to Python 2.3, if the new interpreter is compiled with a new version of the underlying BerkeleyDB library, you will almost certainly have to convert your database files to the new version. You can do this fairly easily with the new scripts `db2pickle.py` and `pickle2db.py` which you will find in the distribution's `Tools/scripts` directory. If you've already been using the PyBSDDDB package and importing it as `bsddb3`, you will have to change your import statements to import it as `bsddb`.
• The new `bz2` module is an interface to the bz2 data compression library. bz2-compressed data is usually smaller than corresponding `zlib`-compressed data. (Contributed by Gustavo Niemeyer.)

• A set of standard date/time types has been added in the new `datetime` module. See the following section for more details.

• The Distutils Extension class now supports an extra constructor argument named `depends` for listing additional source files that an extension depends on. This lets Distutils recompile the module if any of the dependency files are modified. For example, if `sampmodule.c` includes the header file `sample.h`, you would create the Extension object like this:

```python
ext = Extension("samp",
               sources=["sampmodule.c"],
               depends=["sample.h"])
```

Modifying `sample.h` would then cause the module to be recompiled. (Contributed by Jeremy Hylton.)

• Other minor changes to Distutils: it now checks for the `CC`, `CFLAGS`, `CPP`, `LDFLAGS`, and `CPPFLAGS` environment variables, using them to override the settings in Python’s configuration (contributed by Robert Weber).

• Previously the `doctest` module would only search the docstrings of public methods and functions for test cases, but it now also examines private ones as well. The `DocTestSuite()` function creates a `unittest.TestSuite` object from a set of `doctest` tests.
• The new gc.get_referents(object) function returns a list of all the objects referenced by object.

• The getopt module gained a new function, gnu_getopt(), that supports the same arguments as the existing getopt() function but uses GNU-style scanning mode. The existing getopt() stops processing options as soon as a non-option argument is encountered, but in GNU-style mode processing continues, meaning that options and arguments can be mixed. For example:

```python
>>> import getopt
>>> getopt.getopt([('-f', 'filename'), ('output', '-')], ['-f', 'filename'])
>>> getopt.gnu_getopt([('-f', 'filename'), ('output', '-')], ['-f', 'filename'])
```

(Contributed by Peter Åstrand.)

• The grp, pwd, and resource modules now return enhanced tuples:

```python
>>> import grp
>>> g = grp.getgrnam('amk')
>>> g.gr_name, g.gr_gid
('amk', 500)
```

• The gzip module can now handle files exceeding 2 GiB.

• The new heapq module contains an implementation of a heap queue algorithm. A heap is an array-like data structure that keeps items in a partially sorted order such that, for every index $k$, $\text{heap}[k] \leq \text{heap}[2k+1]$ and $\text{heap}[k] \leq \text{heap}[2k+2]$. This makes it quick to remove the smallest item, and inserting a
new item while maintaining the heap property is $O(\log n)$. (See [https://xlinux.nist.gov/dads//HTML/priorityque.html](https://xlinux.nist.gov/dads//HTML/priorityque.html) for more information about the priority queue data structure.)

The `heapq` module provides `heappush()` and `heappop()` functions for adding and removing items while maintaining the heap property on top of some other mutable Python sequence type. Here's an example that uses a Python list:

```python
>>> import heapq
>>> heap = []
>>> for item in [3, 7, 5, 11, 1]:
...    heapq.heappush(heap, item)
...    
>>> heap
[1, 3, 5, 11, 7]
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
3
>>> heap
[5, 7, 11]
```

(Contributed by Kevin O'Connor.)

- The IDLE integrated development environment has been updated using the code from the IDLEfork project ([http://idlefork.sourceforge.net](http://idlefork.sourceforge.net)). The most notable feature is that the code being developed is now executed in a subprocess, meaning that there's no longer any need for manual `reload()` operations. IDLE's core code has been incorporated into the standard library as the `idlelib` package.

- The `imaplib` module now supports IMAP over SSL. (Contributed
The *itertools* contains a number of useful functions for use with iterators, inspired by various functions provided by the ML and Haskell languages. For example, *itertools.ifilter(predicate, iterator)* returns all elements in the iterator for which the function *predicate()* returns *True*, and *itertools.repeat(obj, N)* returns *obj* *N* times. There are a number of other functions in the module; see the package’s reference documentation for details. (Contributed by Raymond Hettinger.)

Two new functions in the *math* module, *degrees(rads)* and *radians(degs)*, convert between radians and degrees. Other functions in the *math* module such as *math.sin()* and *math.cos()* have always required input values measured in radians. Also, an optional *base* argument was added to *math.log()* to make it easier to compute logarithms for bases other than *e* and 10. (Contributed by Raymond Hettinger.)

Several new POSIX functions (*getpgid()*, *killpg()*, *lchown()*, *loadavg()*, *major()*, *makedev()*, *minor()*, and *mknod()*) were added to the *posix* module that underlies the *os* module. (Contributed by Gustavo Niemeyer, Geert Jansen, and Denis S. Otkidach.)

In the *os* module, the *stat()* family of functions can now report fractions of a second in a timestamp. Such time stamps are represented as floats, similar to the value returned by *time.time()*.

During testing, it was found that some applications will break if
time stamps are floats. For compatibility, when using the tuple interface of the stat_result time stamps will be represented as integers. When using named fields (a feature first introduced in Python 2.2), time stamps are still represented as integers, unless os.stat_float_times() is invoked to enable float return values:

```
>>> os.stat("/tmp").st_mtime
1034791200
>>> os.stat_float_times(True)
>>> os.stat("/tmp").st_mtime
1034791200.6335014
```

In Python 2.4, the default will change to always returning floats.

Application developers should enable this feature only if all their libraries work properly when confronted with floating point time stamps, or if they use the tuple API. If used, the feature should be activated on an application level instead of trying to enable it on a per-use basis.

- The optparse module contains a new parser for command-line arguments that can convert option values to a particular Python type and will automatically generate a usage message. See the following section for more details.

- The old and never-documented linuxaudiodev module has been deprecated, and a new version named ossaudiodev has been added. The module was renamed because the OSS sound drivers can be used on platforms other than Linux, and the interface has also been tidied and brought up to date in various ways. (Contributed by Greg Ward and Nicholas FitzRoy-Dale.)
• The new **platform** module contains a number of functions that try to determine various properties of the platform you're running on. There are functions for getting the architecture, CPU type, the Windows OS version, and even the Linux distribution version. (Contributed by Marc-André Lemburg.)

• The parser objects provided by the pyexpat module can now optionally buffer character data, resulting in fewer calls to your character data handler and therefore faster performance. Setting the parser object's buffer_text attribute to **True** will enable buffering.

• The **sample(population, k)** function was added to the **random** module. **population** is a sequence or xrange object containing the elements of a population, and sample() chooses **k** elements from the population without replacing chosen elements. **k** can be any value up to **len(population)**. For example:

```python
>>> days = [ 'Mo', 'Tu', 'We', 'Th', 'Fr', 'St', 'Sn' ]
>>> random.sample(days, 3)  # Choose 3 elements from the population
[ 'St', 'Sn', 'Th' ]
>>> random.sample(days, 7)  # Choose 7 elements
[ 'Tu', 'Th', 'Mo', 'We', 'St', 'Fr', 'Sn' ]
>>> random.sample(days, 7)  # Choose 7 again
[ 'We', 'Mo', 'Sn', 'Fr', 'Tu', 'St', 'Th' ]
>>> random.sample(days, 8)  # Can't choose eight elements
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    File "random.py", line 414, in sample
      raise ValueError, "sample larger than population"
ValueError: sample larger than population
>>> random.sample(xrange(1,10000,2), 10)  # Choose ten odd numbers
[3407, 3805, 1505, 7023, 2401, 2267, 9733, 3151, 8385, 9195]
```
The `random` module now uses a new algorithm, the Mersenne Twister, implemented in C. It's faster and more extensively studied than the previous algorithm.

(All changes contributed by Raymond Hettinger.)

- The `readline` module also gained a number of new functions: `get_history_item()`, `get_current_history_length()`, and `redisplay()`.

- The `reexec` and Bastion modules have been declared dead, and attempts to import them will fail with a `RuntimeError`. New-style classes provide new ways to break out of the restricted execution environment provided by `reexec`, and no one has interest in fixing them or time to do so. If you have applications using `reexec`, rewrite them to use something else.

  (Sticking with Python 2.2 or 2.1 will not make your applications any safer because there are known bugs in the `reexec` module in those versions. To repeat: if you're using `reexec`, stop using it immediately.)

- The `rotor` module has been deprecated because the algorithm it uses for encryption is not believed to be secure. If you need encryption, use one of the several AES Python modules that are available separately.

- The `shutil` module gained a `move(src, dest)` function that recursively moves a file or directory to a new location.

- Support for more advanced POSIX signal handling was added to the `signal` but then removed again as it proved impossible to
make it work reliably across platforms.

- The **socket** module now supports timeouts. You can call the `settimeout(t)` method on a socket object to set a timeout of \( t \) seconds. Subsequent socket operations that take longer than \( t \) seconds to complete will abort and raise a `socket.timeout` exception.

The original timeout implementation was by Tim O’Malley. Michael Gilfix integrated it into the Python **socket** module and shepherded it through a lengthy review. After the code was checked in, Guido van Rossum rewrote parts of it. (This is a good example of a collaborative development process in action.)

- On Windows, the **socket** module now ships with Secure Sockets Layer (SSL) support.

- The value of the C `PYTHON_API_VERSION` macro is now exposed at the Python level as `sys.api_version`. The current exception can be cleared by calling the new `sys.exc_clear()` function.

- The new **tarfile** module allows reading from and writing to tar-format archive files. (Contributed by Lars Gustäbel.)

- The new **textwrap** module contains functions for wrapping strings containing paragraphs of text. The `wrap(text, width)` function takes a string and returns a list containing the text split into lines of no more than the chosen width. The `fill(text, width)` function returns a single string, reformatted to fit into lines no longer than the chosen width. (As you can guess, `fill()` is built on top of `wrap()`.) For example:
>>> import textwrap
>>> paragraph = "Not a whit, we defy augury: ...
>>> textwrap.wrap(paragraph, 60)
["Not a whit, we defy augury: there's a special 
"the fall of a sparrow. If it be now, 'tis not ...
>>> print(textwrap.fill(paragraph, 35))
Not a whit, we defy augury: there's a special providence in the fall of a sparrow. If it be now, 'tis not to come; if it be not to come, it will be now; if it be not now, yet it will come: the readiness is all.

The module also contains a TextWrapper class that actually implements the text wrapping strategy. Both the TextWrapper class and the wrap() and fill() functions support a number of additional keyword arguments for fine-tuning the formatting; consult the module's documentation for details. (Contributed by Greg Ward.)

- The thread and threading modules now have companion modules, dummy_thread and dummy_threading, that provide a do-nothing implementation of the thread module's interface for platforms where threads are not supported. The intention is to simplify thread-aware modules (ones that don't rely on threads to run) by putting the following code at the top:

```python
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```
In this example, _threading is used as the module name to make it clear that the module being used is not necessarily the actual threading module. Code can call functions and use classes in _threading whether or not threads are supported, avoiding an if statement and making the code slightly clearer. This module will not magically make multithreaded code run without threads; code that waits for another thread to return or to do something will simply hang forever.

- The time module's strftime() function has long been an annoyance because it uses the platform C library's strftime() implementation, and different platforms sometimes have odd bugs. Brett Cannon contributed a portable implementation that's written in pure Python and should behave identically on all platforms.

- The new timeit module helps measure how long snippets of Python code take to execute. The timeit.py file can be run directly from the command line, or the module's Timer class can be imported and used directly. Here's a short example that figures out whether it's faster to convert an 8-bit string to Unicode by appending an empty Unicode string to it or by using the unicode() function:

```python
import timeit

timer1 = timeit.Timer('unicode("abc")')
timer2 = timeit.Timer('"abc" + u""'

# Run three trials
print timer1.repeat(repeat=3, number=100000)
print timer2.repeat(repeat=3, number=100000)
```
The Tix module has received various bug fixes and updates for the current version of the Tix package.

The Tkinter module now works with a thread-enabled version of Tcl. Tcl's threading model requires that widgets only be accessed from the thread in which they're created; accesses from another thread can cause Tcl to panic. For certain Tcl interfaces, Tkinter will now automatically avoid this when a widget is accessed from a different thread by marshalling a command, passing it to the correct thread, and waiting for the results. Other interfaces can't be handled automatically but Tkinter will now raise an exception on such an access so that you can at least find out about the problem. See https://mail.python.org/pipermail/python-dev/2002-December/031107.html for a more detailed explanation of this change. (Implemented by Martin von Löwis.)

Calling Tcl methods through _tkinter no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a _tkinter.Tcl_Obj object if no Python equivalent exists. This behavior can be controlled through the wantobjects() method of tkapp objects.

When using _tkinter through the Tkinter module (as most Tkinter applications will), this feature is always activated. It should not cause compatibility problems, since Tkinter would always convert string results to Python types where possible.
If any incompatibilities are found, the old behavior can be restored by setting the wantobjects variable in the Tkinter module to false before creating the first tkapp object.

```python
import Tkinter
Tkinter.wantobjects = 0
```

Any breakage caused by this change should be reported as a bug.

- The UserDict module has a new DictMixin class which defines all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries, such as the classes in the shelve module.

Adding the mix-in as a superclass provides the full dictionary interface whenever the class defines `__getitem__()`, `__setitem__()`, `__delitem__()`, and `keys()`. For example:

```python
>>> import UserDict
>>> class SeqDict(UserDict.DictMixin):
...    """Dictionary lookalike implemented with lists."
...    def __init__(self):
...        self.keylist = []
...        self.valuelist = []
...    def __getitem__(self, key):
...        try:
...            i = self.keylist.index(key)
...        except ValueError:
...            raise KeyError
...        return self.valuelist[i]
...    def __setitem__(self, key, value):
...        try:
```
... i = self.keylist.index(key)
... self.valuelist[i] = value
...
except ValueError:
... self.keylist.append(key)
... self.valuelist.append(value)
...

def __delitem__(self, key):
... try:
... i = self.keylist.index(key)
...
except ValueError:
... raise KeyError
...
self.keylist.pop(i)
...
self.valuelist.pop(i)
...

def keys(self):
... return list(self.keylist)
...

>>> s = SeqDict()
>>> dir(s)  # See that other dictionary methods
['__cmp__', '__contains__', '__delitem__', '__doc__', '__init__', '__iter__', '__len__', '__module__', '__setitem__', 'clear', 'get', 'has_key', 'items', 'iterkeys', 'itervalues', 'keylist', 'keys', 'pop', 'setdefault', 'update', 'valuelist', 'values']

(Contributed by Raymond Hettinger.)

- The DOM implementation in xml.dom.minidom can now generate XML output in a particular encoding by providing an optional encoding argument to the toxml() and toprettyxml() methods of DOM nodes.

- The xmlrpclib module now supports an XML-RPC extension for handling nil data values such as Python’s None. Nil values are always supported on unmarshalling an XML-RPC response. To generate requests containing None, you must supply a true value
for the `allow_none` parameter when creating a Marshaller instance.

- The new DocXMLRPCServer module allows writing self-documenting XML-RPC servers. Run it in demo mode (as a program) to see it in action. Pointing the Web browser to the RPC server produces pydoc-style documentation; pointing xmlrpclib to the server allows invoking the actual methods. (Contributed by Brian Quinlan.)

- Support for internationalized domain names (RFCs 3454, 3490, 3491, and 3492) has been added. The “idna” encoding can be used to convert between a Unicode domain name and the ASCII-compatible encoding (ACE) of that name.

```python
```

The `socket` module has also been extended to transparently convert Unicode hostnames to the ACE version before passing them to the C library. Modules that deal with hostnames such as `httpplib` and `ftplib`) also support Unicode host names; `httpplib` also sends HTTP Host headers using the ACE version of the domain name. `urllib` supports Unicode URLs with non-ASCII host names as long as the path part of the URL is ASCII only.

To implement this change, the `stringprep` module, the `mkstringprep` tool and the `punycode` encoding have been added.

**Date/Time Type**
Date and time types suitable for expressing timestamps were added as the \texttt{datetime} module. The types don't support different calendars or many fancy features, and just stick to the basics of representing time.

The three primary types are: \texttt{date}, representing a day, month, and year; \texttt{time}, consisting of hour, minute, and second; and \texttt{datetime}, which contains all the attributes of both \texttt{date} and \texttt{time}. There's also a \texttt{timedelta} class representing differences between two points in time, and time zone logic is implemented by classes inheriting from the abstract \texttt{tzinfo} class.

You can create instances of \texttt{date} and \texttt{time} by either supplying keyword arguments to the appropriate constructor, e.g. `datetime.date(year=1972, month=10, day=15)`, or by using one of a number of class methods. For example, the \texttt{date.today()} class method returns the current local date.

Once created, instances of the date/time classes are all immutable. There are a number of methods for producing formatted strings from objects:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.isoformat()
'2002-12-30T21:27:03.994956'
>>> now.ctime()  # Only available on date, datetime
'Mon Dec 30 21:27:03 2002'
>>> now.strftime('%Y %d %b')
'2002 30 Dec'
```

The \texttt{replace()} method allows modifying one or more fields of a
date or `datetime` instance, returning a new instance:

```python
>>> d = datetime.datetime.now()
>>> d
datetime.datetime(2002, 12, 30, 22, 15, 38, 827738)
>>> d.replace(year=2001, hour = 12)
datetime.datetime(2001, 12, 30, 12, 15, 38, 827738)
```n
Instances can be compared, hashed, and converted to strings (the result is the same as that of `isoformat()`). `date` and `datetime` instances can be subtracted from each other, and added to `timedelta` instances. The largest missing feature is that there's no standard library support for parsing strings and getting back a date or `datetime`.

For more information, refer to the module's reference documentation. (Contributed by Tim Peters.)

**The optparse Module**

The `getopt` module provides simple parsing of command-line arguments. The new `optparse` module (originally named Optik) provides more elaborate command-line parsing that follows the Unix conventions, automatically creates the output for `--help`, and can perform different actions for different options.

You start by creating an instance of `OptionParser` and telling it what your program's options are.

```python
import sys
from optparse import OptionParser
```
op = OptionParser()
op.add_option('-i', '--input',
    action='store', type='string', dest='input',
    help='set input filename')

op.add_option('-l', '--length',
    action='store', type='int', dest='length',
    help='set maximum length of output')

Parsing a command line is then done by calling the parse_args() method.

```python
options, args = op.parse_args(sys.argv[1:])
print options
print args
```

This returns an object containing all of the option values, and a list of strings containing the remaining arguments.

Invoking the script with the various arguments now works as you'd expect it to. Note that the length argument is automatically converted to an integer.

```bash
$ ./python opt.py -i data arg1
<Values at 0x400cad4c: {'input': 'data', 'length': None, 'arg1': []}>
$ ./python opt.py --input=data --length=4
<Values at 0x400cad2c: {'input': 'data', 'length': 4, 'arg1': []}>
```

The help message is automatically generated for you:

```bash
$ ./python opt.py --help
usage: opt.py [options]
```
options:
  -h, --help                      show this help message and exit
  -i INPUT, --input=INPUT          set input filename
  -l LENGTH, --length=LENGTH      set maximum length of output

See the module's documentation for more details.

Optik was written by Greg Ward, with suggestions from the readers of the Getopt SIG.
Pymalloc: A Specialized Object Allocator

Pymalloc, a specialized object allocator written by Vladimir Marangozov, was a feature added to Python 2.1. Pymalloc is intended to be faster than the system malloc() and to have less memory overhead for allocation patterns typical of Python programs. The allocator uses C's malloc() function to get large pools of memory and then fulfills smaller memory requests from these pools.

In 2.1 and 2.2, pymalloc was an experimental feature and wasn’t enabled by default; you had to explicitly enable it when compiling Python by providing the --with-pymalloc option to the configure script. In 2.3, pymalloc has had further enhancements and is now enabled by default; you’ll have to supply --without-pymalloc to disable it.

This change is transparent to code written in Python; however, pymalloc may expose bugs in C extensions. Authors of C extension modules should test their code with pymalloc enabled, because some incorrect code may cause core dumps at runtime.

There’s one particularly common error that causes problems. There are a number of memory allocation functions in Python’s C API that have previously just been aliases for the C library’s malloc() and free(), meaning that if you accidentally called mismatched functions the error wouldn’t be noticeable. When the object allocator is enabled, these functions aren’t aliases of malloc() and free() any more, and calling the wrong function to free memory may get you a core dump. For example, if memory was allocated using PyObject_Malloc(), it has to be freed using PyObject_Free(), not free(). A few modules included with Python fell afoul of this and
had to be fixed; doubtless there are more third-party modules that will have the same problem.

As part of this change, the confusing multiple interfaces for allocating memory have been consolidated down into two API families. Memory allocated with one family must not be manipulated with functions from the other family. There is one family for allocating chunks of memory and another family of functions specifically for allocating Python objects.

- To allocate and free an undistinguished chunk of memory use the “raw memory” family: `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()`.
- The “object memory” family is the interface to the pymalloc facility described above and is biased towards a large number of “small” allocations: `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()`.
- To allocate and free Python objects, use the “object” family `PyObject_New()`, `PyObject_NewVar()`, and `PyObject_Del()`.

Thanks to lots of work by Tim Peters, pymalloc in 2.3 also provides debugging features to catch memory overwrites and doubled frees in both extension modules and in the interpreter itself. To enable this support, compile a debugging version of the Python interpreter by running `configure` with `--with-pydebug`.

To aid extension writers, a header file `Misc/pymemcompat.h` is distributed with the source to Python 2.3 that allows Python extensions to use the 2.3 interfaces to memory allocation while compiling against any version of Python since 1.5.2. You would copy the file from Python’s source distribution and bundle it with the source of your extension.
See also:

https://hg.python.org/cpython/file/default/Objects/obmalloc.c

For the full details of the pymalloc implementation, see the comments at the top of the file Objects/obmalloc.c in the Python source code. The above link points to the file within the python.org SVN browser.
Build and C API Changes

Changes to Python’s build process and to the C API include:

- The cycle detection implementation used by the garbage collection has proven to be stable, so it’s now been made mandatory. You can no longer compile Python without it, and the --with-cycle-gc switch to configure has been removed.
- Python can now optionally be built as a shared library (libpython2.3.so) by supplying --enable-shared when running Python’s configure script. (Contributed by Ondrej Palkovsky.)
- The DL_EXPORT and DL_IMPORT macros are now deprecated. Initialization functions for Python extension modules should now be declared using the new macro PyMODINIT_FUNC, while the Python core will generally use the PyAPI_FUNC and PyAPI_DATA macros.
- The interpreter can be compiled without any docstrings for the built-in functions and modules by supplying --without-docstrings to the configure script. This makes the Python executable about 10% smaller, but will also mean that you can’t get help for Python’s built-ins. (Contributed by Gustavo Niemeyer.)
- The PyArg_NoArgs() macro is now deprecated, and code that uses it should be changed. For Python 2.2 and later, the method definition table can specify the METH_NOARGS flag, signalling that there are no arguments, and the argument checking can then be removed. If compatibility with pre-2.2 versions of Python is important, the code could use PyArg_ParseTuple(args, "") instead, but this will be slower than using METH_NOARGS.
- PyArg_ParseTuple() accepts new format characters for various sizes of unsigned integers: B for unsigned char, H for unsigned short int, I for unsigned int, and K for
• A new function, `PyObject_DelItemString(mapping, char *key)` was added as shorthand for `PyObject_DelItem(mapping, PyString_New(key))`.

• File objects now manage their internal string buffer differently, increasing it exponentially when needed. This results in the benchmark tests in `Lib/test/test_bufio.py` speeding up considerably (from 57 seconds to 1.7 seconds, according to one measurement).

• It's now possible to define class and static methods for a C extension type by setting either the `METH_CLASS` or `METH_STATIC` flags in a method's `PyMethodDef` structure.

• Python now includes a copy of the Expat XML parser's source code, removing any dependence on a system version or local installation of Expat.

• If you dynamically allocate type objects in your extension, you should be aware of a change in the rules relating to the `__module__` and `__name__` attributes. In summary, you will want to ensure the type's dictionary contains a `'__module__'` key; making the module name the part of the type name leading up to the final period will no longer have the desired effect. For more detail, read the API reference documentation or the source.

**Port-Specific Changes**

Support for a port to IBM’s OS/2 using the EMX runtime environment was merged into the main Python source tree. EMX is a POSIX emulation layer over the OS/2 system APIs. The Python port for EMX tries to support all the POSIX-like capability exposed by the EMX runtime, and mostly succeeds; `fork()` and `fcntl()` are restricted by the limitations of the underlying emulation layer. The standard OS/2 port, which uses IBM’s Visual Age compiler, also gained support for
case-sensitive import semantics as part of the integration of the EMX port into CVS. (Contributed by Andrew MacIntyre.)

On MacOS, most toolbox modules have been weaklinked to improve backward compatibility. This means that modules will no longer fail to load if a single routine is missing on the current OS version. Instead calling the missing routine will raise an exception. (Contributed by Jack Jansen.)

The RPM spec files, found in the Misc/RPM/ directory in the Python source distribution, were updated for 2.3. (Contributed by Sean Reifschneider.)

Other new platforms now supported by Python include AtheOS (http://www.atheos.cx/), GNU/Hurd, and OpenVMS.
Other Changes and Fixes

As usual, there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 523 patches applied and 514 bugs fixed between Python 2.2 and 2.3. Both figures are likely to be underestimates.

Some of the more notable changes are:

- If the PYTHONINSPECT environment variable is set, the Python interpreter will enter the interactive prompt after running a Python program, as if Python had been invoked with the \texttt{-i} option. The environment variable can be set before running the Python interpreter, or it can be set by the Python program as part of its execution.

- The \texttt{regrtest.py} script now provides a way to allow “all resources except \texttt{foo}.” A resource name passed to the \texttt{-u} option can now be prefixed with a hyphen (\texttt{'-'} ) to mean “remove this resource.” For example, the option \texttt{'-uall,-bsddb'} could be used to enable the use of all resources except bsddb.

- The tools used to build the documentation now work under Cygwin as well as Unix.

- The \texttt{SET_LINENO} opcode has been removed. Back in the mists of time, this opcode was needed to produce line numbers in tracebacks and support trace functions (for, e.g.,\texttt{pdb}). Since Python 1.5, the line numbers in tracebacks have been computed using a different mechanism that works with “python \texttt{-O}”. For Python 2.3 Michael Hudson implemented a similar scheme to
determine when to call the trace function, removing the need for \texttt{SET\_LINENO} entirely.

It would be difficult to detect any resulting difference from Python code, apart from a slight speed up when Python is run without \texttt{-O}.

C extensions that access the \texttt{f\_lineno} field of frame objects should instead call \texttt{PyCode\_Addr2Line(f->f\_code, f->f\_lasti)}. This will have the added effect of making the code work as desired under “python -O” in earlier versions of Python.

A nifty new feature is that trace functions can now assign to the \texttt{f\_lineno} attribute of frame objects, changing the line that will be executed next. A \texttt{jump} command has been added to the \texttt{pdb} debugger taking advantage of this new feature. (Implemented by Richie Hindle.)
Porting to Python 2.3

This section lists previously described changes that may require changes to your code:

- **yield** is now always a keyword; if it’s used as a variable name in your code, a different name must be chosen.

- For strings \(X\) and \(Y\), \(X \text{ in } Y\) now works if \(X\) is more than one character long.

- The \texttt{int()} type constructor will now return a long integer instead of raising an \texttt{OverflowError} when a string or floating-point number is too large to fit into an integer.

- If you have Unicode strings that contain 8-bit characters, you must declare the file’s encoding (UTF-8, Latin-1, or whatever) by adding a comment to the top of the file. See section \texttt{PEP 263: Source Code Encodings} for more information.

- Calling Tcl methods through \_tkinter no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a \_tkinter.Tcl_Obj object if no Python equivalent exists.

- Large octal and hex literals such as \(0xfffffffff\) now trigger a \texttt{FutureWarning}. Currently they’re stored as 32-bit numbers and result in a negative value, but in Python 2.4 they’ll become positive long integers.

There are a few ways to fix this warning. If you really need a positive number, just add an \texttt{L} to the end of the literal. If you’re
trying to get a 32-bit integer with low bits set and have previously used an expression such as ~(1 << 31), it's probably clearest to start with all bits set and clear the desired upper bits. For example, to clear just the top bit (bit 31), you could write 0xffffffffL &~(1L << 31).

- You can no longer disable assertions by assigning to __debug__.

- The Distutils setup() function has gained various new keyword arguments such as depends. Old versions of the Distutils will abort if passed unknown keywords. A solution is to check for the presence of the new get_distutil_options() function in your setup.py and only uses the new keywords with a version of the Distutils that supports them:

```python
from distutils import core
kw = {'sources': 'foo.c', ...}
if hasattr(core, 'get_distutil_options'):
    kw['depends'] = ['foo.h']
ext = Extension(**kw)
```

- Using None as a variable name will now result in a SyntaxWarning warning.

- Names of extension types defined by the modules included with Python now contain the module and a '.' in front of the type name.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Jeff Bauer, Simon Brunning, Brett Cannon, Michael Chermside, Andrew Dalke, Scott David Daniels, Fred L. Drake, Jr., David Fraser, Kelly Gerber, Raymond Hettinger, Michael Hudson, Chris Lambert, Detlef Lannert, Martin von Löwis, Andrew MacIntyre, Lalo Martins, Chad Netzer, Gustavo Niemeyer, Neal Norwitz, Hans Nowak, Chris Reedy, Francesco Ricciardi, Vinay Sajip, Neil Schemenauer, Roman Suzi, Jason Tishler, Just van Rossum.
What’s New in Python 2.2

Author: A.M. Kuchling
Introduction

This article explains the new features in Python 2.2.2, released on October 14, 2002. Python 2.2.2 is a bugfix release of Python 2.2, originally released on December 21, 2001.

Python 2.2 can be thought of as the “cleanup release”. There are some features such as generators and iterators that are completely new, but most of the changes, significant and far-reaching though they may be, are aimed at cleaning up irregularities and dark corners of the language design.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.2, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature.
PEPs 252 and 253: Type and Class Changes

The largest and most far-reaching changes in Python 2.2 are to Python’s model of objects and classes. The changes should be backward compatible, so it’s likely that your code will continue to run unchanged, but the changes provide some amazing new capabilities. Before beginning this, the longest and most complicated section of this article, I’ll provide an overview of the changes and offer some comments.

A long time ago I wrote a Web page listing flaws in Python’s design. One of the most significant flaws was that it’s impossible to subclass Python types implemented in C. In particular, it’s not possible to subclass built-in types, so you can’t just subclass, say, lists in order to add a single useful method to them. The UserList module provides a class that supports all of the methods of lists and that can be subclassed further, but there’s lots of C code that expects a regular Python list and won’t accept a UserList instance.

Python 2.2 fixes this, and in the process adds some exciting new capabilities. A brief summary:

- You can subclass built-in types such as lists and even integers, and your subclasses should work in every place that requires the original type.
- It’s now possible to define static and class methods, in addition to the instance methods available in previous versions of Python.
- It’s also possible to automatically call methods on accessing or setting an instance attribute by using a new mechanism called properties. Many uses of __getattr__() can be rewritten to use properties instead, making the resulting code simpler and
faster. As a small side benefit, attributes can now have docstrings, too.

- The list of legal attributes for an instance can be limited to a particular set using *slots*, making it possible to safeguard against typos and perhaps make more optimizations possible in future versions of Python.

Some users have voiced concern about all these changes. Sure, they say, the new features are neat and lend themselves to all sorts of tricks that weren’t possible in previous versions of Python, but they also make the language more complicated. Some people have said that they’ve always recommended Python for its simplicity, and feel that its simplicity is being lost.

Personally, I think there’s no need to worry. Many of the new features are quite esoteric, and you can write a lot of Python code without ever needed to be aware of them. Writing a simple class is no more difficult than it ever was, so you don’t need to bother learning or teaching them unless they’re actually needed. Some very complicated tasks that were previously only possible from C will now be possible in pure Python, and to my mind that’s all for the better.

I’m not going to attempt to cover every single corner case and small change that were required to make the new features work. Instead this section will paint only the broad strokes. See section Related Links, “Related Links”, for further sources of information about Python 2.2’s new object model.

**Old and New Classes**

First, you should know that Python 2.2 really has two kinds of classes: classic or old-style classes, and new-style classes. The old-style class model is exactly the same as the class model in earlier versions of
Python. All the new features described in this section apply only to new-style classes. This divergence isn’t intended to last forever; eventually old-style classes will be dropped, possibly in Python 3.0.

So how do you define a new-style class? You do it by subclassing an existing new-style class. Most of Python’s built-in types, such as integers, lists, dictionaries, and even files, are new-style classes now. A new-style class named `object`, the base class for all built-in types, has also been added so if no built-in type is suitable, you can just subclass `object`:

```python
class C(object):
    def __init__(self):
        ...
```

This means that `class` statements that don’t have any base classes are always classic classes in Python 2.2. (Actually you can also change this by setting a module-level variable named `__metaclass__` — see PEP 253 for the details — but it’s easier to just subclass `object`.)

The type objects for the built-in types are available as built-ins, named using a clever trick. Python has always had built-in functions named `int()`, `float()`, and `str()`. In 2.2, they aren’t functions any more, but type objects that behave as factories when called.

```python
>>> int
<type 'int'>
>>> int('123')
123
```

To make the set of types complete, new type objects such as `dict()`
and file() have been added. Here’s a more interesting example, adding a lock() method to file objects:

```python
class LockableFile(file):
    def lock(self, operation, length=0, start=0, whence=0):
        import fcntl
        return fcntl.lockf(self.fileno(), operation, length, start, whence)
```

The now-obsolete posixfile module contained a class that emulated all of a file object’s methods and also added a lock() method, but this class couldn’t be passed to internal functions that expected a built-in file, something which is possible with our new LockableFile.

## Descriptors

In previous versions of Python, there was no consistent way to discover what attributes and methods were supported by an object. There were some informal conventions, such as defining __members__ and __methods__ attributes that were lists of names, but often the author of an extension type or a class wouldn’t bother to define them. You could fall back on inspecting the __dict__ of an object, but when class inheritance or an arbitrary __getattr__() hook were in use this could still be inaccurate.

The one big idea underlying the new class model is that an API for describing the attributes of an object using descriptors has been formalized. Descriptors specify the value of an attribute, stating whether it’s a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic
Attribute descriptors are objects that live inside class objects, and have a few attributes of their own:

- **__name__** is the attribute’s name.
- **__doc__** is the attribute’s docstring.
- **__get__(object)** is a method that retrieves the attribute value from `object`.
- **__set__(object, value)** sets the attribute on `object` to `value`.
- **__delete__(object, value)** deletes the `value` attribute of `object`.

For example, when you write `obj.x`, the steps that Python actually performs are:

```python
descriptor = obj.__class__.x
descriptor.__get__(obj)
```

For methods, `descriptor.__get__()` returns a temporary object that's callable, and wraps up the instance and the method to be called on it. This is also why static methods and class methods are now possible; they have descriptors that wrap up just the method, or the method and the class. As a brief explanation of these new kinds of methods, static methods aren't passed the instance, and therefore resemble regular functions. Class methods are passed the class of the object, but not the object itself. Static and class methods are defined like this:

```python
class C(object):
    def f(arg1, arg2):
        ...
    f = staticmethod(f)
```
def g(cls, arg1, arg2):
...
g = classmethod(g)

The `staticmethod()` function takes the function $f()$, and returns it wrapped up in a descriptor so it can be stored in the class object. You might expect there to be special syntax for creating such methods (`def static f, defstatic f()`), or something like that) but no such syntax has been defined yet; that's been left for future versions of Python.

More new features, such as slots and properties, are also implemented as new kinds of descriptors, and it's not difficult to write a descriptor class that does something novel. For example, it would be possible to write a descriptor class that made it possible to write Eiffel-style preconditions and postconditions for a method. A class that used this feature might be defined like this:

```python
from eiffel import eiffelmethod

class C(object):
    def f(self, arg1, arg2):
        # The actual function
        ...
    def pre_f(self):
        # Check preconditions
        ...
    def post_f(self):
        # Check postconditions
        ...

    f = eiffelmethod(f, pre_f, post_f)
```

Note that a person using the new `eiffelmethod()` doesn't have to
understand anything about descriptors. This is why I think the new features don't increase the basic complexity of the language. There will be a few wizards who need to know about it in order to write eiffelmethod() or the ZODB or whatever, but most users will just write code on top of the resulting libraries and ignore the implementation details.

Multiple Inheritance: The Diamond Rule

Multiple inheritance has also been made more useful through changing the rules under which names are resolved. Consider this set of classes (diagram taken from PEP 253 by Guido van Rossum):

```
class A:
    ^  ^  def save(self): ...
       /    /
      /     /
     /      /
    class B  class C:
        ^  ^  def save(self): ...
           /    /
          /     /
         /      /
        /       /
       /        /
      class D
```

The lookup rule for classic classes is simple but not very smart; the base classes are searched depth-first, going from left to right. A reference to D.save() will search the classes D, B, and then A, where save() would be found and returned. C.save() would never be found at all. This is bad, because if C’s save() method is saving some internal state specific to C, not calling it will result in that state
never getting saved.

New-style classes follow a different algorithm that's a bit more complicated to explain, but does the right thing in this situation. (Note that Python 2.3 changes this algorithm to one that produces the same results in most cases, but produces more useful results for really complicated inheritance graphs.)

1. List all the base classes, following the classic lookup rule and include a class multiple times if it's visited repeatedly. In the above example, the list of visited classes is [D, B, A, C, A].
2. Scan the list for duplicated classes. If any are found, remove all but one occurrence, leaving the last one in the list. In the above example, the list becomes [D, B, C, A] after dropping duplicates.

Following this rule, referring to D.save() will return C.save(), which is the behaviour we're after. This lookup rule is the same as the one followed by Common Lisp. A new built-in function, super(), provides a way to get at a class's superclasses without having to reimplement Python's algorithm. The most commonly used form will be super(class, obj), which returns a bound superclass object (not the actual class object). This form will be used in methods to call a method in the superclass; for example, D's save() method would look like this:

```python
class D (B,C):
    def save (self):
        # Call superclass .save()
        super(D, self).save()
        # Save D's private information here
        ...
```

super() can also return unbound superclass objects when called as
super(class) or super(class1, class2), but this probably won’t often be useful.

**Attribute Access**

A fair number of sophisticated Python classes define hooks for attribute access using `__getattr__()`; most commonly this is done for convenience, to make code more readable by automatically mapping an attribute access such as `obj.parent` into a method call such as `obj.get_parent`. Python 2.2 adds some new ways of controlling attribute access.

First, `__getattr__(attr_name)` is still supported by new-style classes, and nothing about it has changed. As before, it will be called when an attempt is made to access `obj.foo` and no attribute named `foo` is found in the instance's dictionary.

New-style classes also support a new method, `__getattribute__(attr_name)`. The difference between the two methods is that `__getattribute__()` is *always* called whenever any attribute is accessed, while the old `__getattr__()` is only called if `foo` isn’t found in the instance’s dictionary.

However, Python 2.2’s support for `properties` will often be a simpler way to trap attribute references. Writing a `__getattr__()` method is complicated because to avoid recursion you can’t use regular attribute accesses inside them, and instead have to mess around with the contents of `__dict__`. `__getattribute__()` methods also end up being called by Python when it checks for other methods such as `__repr__()` or `__coerce__()`, and so have to be written with this in mind. Finally, calling a function on every attribute access results in a
sizable performance loss.

**property** is a new built-in type that packages up three functions that get, set, or delete an attribute, and a docstring. For example, if you want to define a size attribute that's computed, but also settable, you could write:

```python
class C(object):
    def get_size(self):
        result = ... computation ...
        return result
    def set_size(self, size):
        ... compute something based on the size and set internal state appropriately ...

    size = property(get_size, set_size, None,
                    "Storage size of this instance")
```

That is certainly clearer and easier to write than a pair of `__getattr__()`/`__setattr__()` methods that check for the size attribute and handle it specially while retrieving all other attributes from the instance's `__dict__`. Accesses to size are also the only ones which have to perform the work of calling a function, so references to other attributes run at their usual speed.

Finally, it's possible to constrain the list of attributes that can be referenced on an object using the new `__slots__` class attribute. Python objects are usually very dynamic; at any time it's possible to define a new attribute on an instance by just doing `obj.new_attr=1`. 
A new-style class can define a class attribute named __slots__ to limit the legal attributes to a particular set of names. An example will make this clear:

```python
>>> class C(object):
...     __slots__ = ('template', 'name')
...
>>> obj = C()
>>> print obj.template
None
>>> obj.template = 'Test'
>>> print obj.template
Test
>>> obj.newattr = None
AttributeError: 'C' object has no attribute 'newattr'
```

Note how you get an AttributeError on the attempt to assign to an attribute not listed in __slots__.

**Related Links**

This section has just been a quick overview of the new features, giving enough of an explanation to start you programming, but many details have been simplified or ignored. Where should you go to get a more complete picture?

[https://docs.python.org/dev/howto/descriptor.html](https://docs.python.org/dev/howto/descriptor.html) is a lengthy tutorial introduction to the descriptor features, written by Guido van Rossum. If my description has whetted your appetite, go read this tutorial next, because it goes into much more detail about the new features while still remaining quite easy to read.
Next, there are two relevant PEPs, PEP 252 and PEP 253. PEP 252 is titled “Making Types Look More Like Classes”, and covers the descriptor API. PEP 253 is titled “Subtyping Built-in Types”, and describes the changes to type objects that make it possible to subtype built-in objects. PEP 253 is the more complicated PEP of the two, and at a few points the necessary explanations of types and meta-types may cause your head to explode. Both PEPs were written and implemented by Guido van Rossum, with substantial assistance from the rest of the Zope Corp. team.

Finally, there’s the ultimate authority: the source code. Most of the machinery for the type handling is in Objects/typeobject.c, but you should only resort to it after all other avenues have been exhausted, including posting a question to python-list or python-dev.
PEP 234: Iterators

Another significant addition to 2.2 is an iteration interface at both the C and Python levels. Objects can define how they can be looped over by callers.

In Python versions up to 2.1, the usual way to make for item in obj work is to define a `__getitem__()` method that looks something like this:

```python
def __getitem__(self, index):
    return <next item>
```

`__getitem__()` is more properly used to define an indexing operation on an object so that you can write `obj[5]` to retrieve the sixth element. It’s a bit misleading when you’re using this only to support for loops. Consider some file-like object that wants to be looped over; the `index` parameter is essentially meaningless, as the class probably assumes that a series of `__getitem__()` calls will be made with `index` incrementing by one each time. In other words, the presence of the `__getitem__()` method doesn’t mean that using `file[5]` to randomly access the sixth element will work, though it really should.

In Python 2.2, iteration can be implemented separately, and `__getitem__()` methods can be limited to classes that really do support random access. The basic idea of iterators is simple. A new built-in function, `iter(obj)` or `iter(C, sentinel)`, is used to get an iterator. `iter(obj)` returns an iterator for the object `obj`, while `iter(C, sentinel)` returns an iterator that will invoke the callable object `C` until it returns `sentinel` to signal that the iterator is done.
Python classes can define an \_\_iter\_() method, which should create and return a new iterator for the object; if the object is its own iterator, this method can just return \_\_self\_. In particular, iterators will usually be their own iterators. Extension types implemented in C can implement a \_\_iter\_ function in order to return an iterator, and extension types that want to behave as iterators can define a \_\_iter\_next\_ function.

So, after all this, what do iterators actually do? They have one required method, \_\_next\_(), which takes no arguments and returns the next value. When there are no more values to be returned, calling \_\_next\_() should raise the \_\_StopIteration\_ exception.

```
>>> L = [1,2,3]
>>> i = iter(L)
>>> print i
<iterator object at 0x8116870>
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>> 
```

In 2.2, Python’s \_\_for\_ statement no longer expects a sequence; it expects something for which \_\_iter\_() will return an iterator. For backward compatibility and convenience, an iterator is automatically constructed for sequences that don’t implement \_\_iter\_() or a
**tp_iter** slot, so for `i in [1,2,3]` will still work. Wherever the Python interpreter loops over a sequence, it's been changed to use the iterator protocol. This means you can do things like this:

```python
>>> L = [1,2,3]
>>> i = iter(L)
>>> a,b,c = i
>>> a,b,c
(1, 2, 3)
```

Iterator support has been added to some of Python’s basic types. Calling `iter()` on a dictionary will return an iterator which loops over its keys:

```python
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, ...
    'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m: print(key, m[key])
... Mar 3
Feb 2
Aug 8
Sep 9
May 5
Jun 6
Jul 7
Jan 1
Apr 4
Nov 11
Dec 12
Oct 12
```

That's just the default behaviour. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate
iterator. In a minor related change, the `in` operator now works on dictionaries, so `key in dict` is now equivalent to `dict.has_key(key)`.

Files also provide an iterator, which calls the `readline()` method until there are no more lines in the file. This means you can now read each line of a file using code like this:

```python
for line in file:
    # do something for each line
...
```

Note that you can only go forward in an iterator; there’s no way to get the previous element, reset the iterator, or make a copy of it. An iterator object could provide such additional capabilities, but the iterator protocol only requires a `next()` method.

**See also:**

**PEP 234 - Iterators**

Written by Ka-Ping Yee and GvR; implemented by the Python Labs crew, mostly by GvR and Tim Peters.
PEP 255: Simple Generators

Generators are another new feature, one that interacts with the introduction of iterators.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren't thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a `yield` statement is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result. Because a new keyword was introduced, generators must be explicitly enabled in a module by including a `from __future__ import generators` statement near the top of the module’s source code. In Python 2.3 this statement will become unnecessary.

When you call a generator function, it doesn’t return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of
i, similar to a return statement. The big difference between yield and a return statement is that on reaching a yield the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's next() method, the function will resume executing immediately after the yield statement. (For complicated reasons, the yield statement isn't allowed inside the try block of a try...finally statement; read PEP 255 for a full explanation of the interaction between yield and exceptions.)

Here's a sample usage of the generate_ints() generator:

```python
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>"", line 1, in ?
  File "<stdin>"", line 2, in generate_ints
StopIteration
```

You could equally write for i in generate_ints(5), or a,b,c = generate_ints(3).

Inside a generator function, the return statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. return with a value, such as return 5, is a syntax error inside a generator
function. The end of the generator’s results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. `Lib/test/test_generators.py` contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in `Lib/test/test_generators.py` produce solutions for the N-Queens problem (placing $N$ queens on an $NxN$ chess board so that no queen threatens another) and the Knight’s Tour (a route that takes a knight to every square of an $NxN$ chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon ([https://www.cs.arizona.edu/icon/](https://www.cs.arizona.edu/icon/)), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon
at https://www.cs.arizona.edu/icon/docs/ipd266.htm gives an idea of what this looks like:

```python
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

In Icon the `find()` function returns the indexes at which the substring “or” is found: 3, 23, 33. In the `if` statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered a new part of the core Python language, but learning or using them isn't compulsory; if they don't solve any problems that you have, feel free to ignore them. One novel feature of Python’s interface as compared to Icon’s is that a generator’s state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See also:

**PEP 255 - Simple Generators**
Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.
PEP 237: Unifying Long Integers and Integers

In recent versions, the distinction between regular integers, which are 32-bit values on most machines, and long integers, which can be of arbitrary size, was becoming an annoyance. For example, on platforms that support files larger than \(2^{32}\) bytes, the `tell()` method of file objects has to return a long integer. However, there were various bits of Python that expected plain integers and would raise an error if a long integer was provided instead. For example, in Python 1.5, only regular integers could be used as a slice index, and `abc'[1L:]` would raise a `TypeError` exception with the message ‘slice index must be int’.

Python 2.2 will shift values from short to long integers as required. The ‘L’ suffix is no longer needed to indicate a long integer literal, as now the compiler will choose the appropriate type. (Using the ‘L’ suffix will be discouraged in future 2.x versions of Python, triggering a warning in Python 2.4, and probably dropped in Python 3.0.) Many operations that used to raise an `OverflowError` will now return a long integer as their result. For example:

```python
>>> 1234567890123
1234567890123L
>>> 2 ** 64
18446744073709551616L
```

In most cases, integers and long integers will now be treated identically. You can still distinguish them with the `type()` built-in function, but that's rarely needed.
See also:

**PEP 237 - Unifying Long Integers and Integers**
Written by Moshe Zadka and Guido van Rossum. Implemented mostly by Guido van Rossum.
PEP 238: Changing the Division Operator

The most controversial change in Python 2.2 heralds the start of an effort to fix an old design flaw that’s been in Python from the beginning. Currently Python's division operator, `/`, behaves like C's division operator when presented with two integer arguments: it returns an integer result that's truncated down when there would be a fractional part. For example, `3/2` is 1, not 1.5, and `(-1)/2` is -1, not -0.5. This means that the results of division can vary unexpectedly depending on the type of the two operands and because Python is dynamically typed, it can be difficult to determine the possible types of the operands.

(The controversy is over whether this is really a design flaw, and whether it's worth breaking existing code to fix this. It's caused endless discussions on python-dev, and in July 2001 erupted into a storm of acidly sarcastic postings on comp.lang.python. I won’t argue for either side here and will stick to describing what's implemented in 2.2. Read PEP 238 for a summary of arguments and counter-arguments.)

Because this change might break code, it's being introduced very gradually. Python 2.2 begins the transition, but the switch won’t be complete until Python 3.0.

First, I'll borrow some terminology from PEP 238. “True division” is the division that most non-programmers are familiar with: `3/2` is 1.5, `1/4` is 0.25, and so forth. “Floor division” is what Python’s `/` operator currently does when given integer operands; the result is the floor of the value returned by true division. “Classic division” is the current mixed behaviour of `/`; it returns the result of floor division when the operands are integers, and returns the result of true division when one
of the operands is a floating-point number.

Here are the changes 2.2 introduces:

- A new operator, `//`, is the floor division operator. (Yes, we know it looks like C++'s comment symbol.) `//` always performs floor division no matter what the types of its operands are, so `1 // 2` is 0 and `1.0 // 2.0` is also 0.0.

  `//` is always available in Python 2.2; you don’t need to enable it using a `__future__` statement.

- By including a `from __future__ import division` in a module, the `/` operator will be changed to return the result of true division, so `1/2` is 0.5. Without the `__future__` statement, `/` still means classic division. The default meaning of `/` will not change until Python 3.0.

- Classes can define methods called `__truediv__()` and `__floordiv__()` to overload the two division operators. At the C level, there are also slots in the `PyNumberMethods` structure so extension types can define the two operators.

- Python 2.2 supports some command-line arguments for testing whether code will work with the changed division semantics. Running python with `-Q warn` will cause a warning to be issued whenever division is applied to two integers. You can use this to find code that’s affected by the change and fix it. By default, Python 2.2 will simply perform classic division without a warning; the warning will be turned on by default in Python 2.3.

See also:
PEP 238 - Changing the Division Operator

Written by Moshe Zadka and Guido van Rossum. Implemented by Guido van Rossum.
Unicode Changes

Python’s Unicode support has been enhanced a bit in 2.2. Unicode strings are usually stored as UCS-2, as 16-bit unsigned integers. Python 2.2 can also be compiled to use UCS-4, 32-bit unsigned integers, as its internal encoding by supplying --enable-unicode=ucs4 to the configure script. (It’s also possible to specify - -disable-unicode to completely disable Unicode support.)

When built to use UCS-4 (a “wide Python”), the interpreter can natively handle Unicode characters from U+000000 to U+110000, so the range of legal values for the unichr() function is expanded accordingly. Using an interpreter compiled to use UCS-2 (a “narrow Python”), values greater than 65535 will still cause unichr() to raise a ValueError exception. This is all described in PEP 261, “Support for ‘wide’ Unicode characters”; consult it for further details.

Another change is simpler to explain. Since their introduction, Unicode strings have supported an encode() method to convert the string to a selected encoding such as UTF-8 or Latin-1. A symmetric decode([*encoding*]) method has been added to 8-bit strings (though not to Unicode strings) in 2.2. decode() assumes that the string is in the specified encoding and decodes it, returning whatever is returned by the codec.

Using this new feature, codecs have been added for tasks not directly related to Unicode. For example, codecs have been added for uu-encoding, MIME’s base64 encoding, and compression with the zlib module:

```python
>>> s = """Here is a lengthy piece of redundant, overly verbose,
```
... and repetitive text.
...

>>> data = s.encode('zlib')

>>> data
'x\x9c\r\xc9\xc1\r\x80 \x10\x04\xc0?Ul...

>>> data.decode('zlib')
'Here is a lengthy piece of redundant, overly verbose, and repetitive text.'

>>> print s.encode('uu')
begin 666 <data>
M2&5R92!I<R!A(&QE;F=T:'D@<&EE8V4@;V8@<F5D=6YD86YT+"!
>=F5R8F]S92P*86YD(')E<&5T:71I=F4@=&5X="X*

end

>>> "sheesh".encode('rot-13')
'furrfu'

To convert a class instance to Unicode, a __unicode__() method can be defined by a class, analogous to __str__().

encode(), decode(), and __unicode__() were implemented by Marc-André Lemburg. The changes to support using UCS-4 internally were implemented by Fredrik Lundh and Martin von Löwis.

See also:

PEP 261 - Support for ‘wide’ Unicode characters
Written by Paul Prescod.
PEP 227: Nested Scopes

In Python 2.1, statically nested scopes were added as an optional feature, to be enabled by a `from __future__ import nested_scopes` directive. In 2.2 nested scopes no longer need to be specially enabled, and are now always present. The rest of this section is a copy of the description of nested scopes from my “What’s New in Python 2.1” document; if you read it when 2.1 came out, you can skip the rest of this section.

The largest change introduced in Python 2.1, and made complete in 2.2, is to Python’s scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn’t match their intuitive expectations. For example, a nested recursive function definition doesn’t work:

```python
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function `g()` will always raise a `NameError` exception, because the binding of the name `g` isn’t in either its local namespace or in the module-level namespace. This isn’t much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` expression clumsier, and this was a problem in practice. In code which uses `lambda` you can often find local variables being copied by passing them as the default values of arguments.
The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.2 is that static scoping has been added to the language to fix this problem. As a first effect, the name=\texttt{name} default argument is now unnecessary in the above example. Put simply, when a given variable \texttt{name} is not assigned a value within a function (by an assignment, or the \texttt{def}, \texttt{class}, or \texttt{import} statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable \texttt{name} is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the \texttt{from module import *} and \texttt{exec} statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that \texttt{from module import *} is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. \texttt{from module import *}
and `exec` make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or `lambda` expressions with free variables, the compiler will flag this by raising a `SyntaxError` exception.

To make the preceding explanation a bit clearer, here’s an example:

```python
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
def g():
    return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by `g()`.

This shouldn’t be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it’s often a sign of a poor design anyway).

See also:

**PEP 227 - Statically Nested Scopes**

Written and implemented by Jeremy Hylton.
New and Improved Modules

- The xmlrpclib module was contributed to the standard library by Fredrik Lundh, providing support for writing XML-RPC clients. XML-RPC is a simple remote procedure call protocol built on top of HTTP and XML. For example, the following snippet retrieves a list of RSS channels from the O'Reilly Network, and then lists the recent headlines for one channel:

```python
import xmlrpclib
channels = s.meerkat.getChannels()
# channels is a list of dictionaries, like this:
# [{'id': 4, 'title': 'Freshmeat Daily News'},
#  {'id': 190, 'title': '32Bits Online'},
#  {'id': 4549, 'title': '3DGamers'}, ...
]

# Get the items for one channel
items = s.meerkat.getItems( {'channel': 4} )
# 'items' is another list of dictionaries, like this:
# [{'link': 'http://freshmeat.net/releases/52719/',
#   'description': 'A utility which converts HTML to XSL FO.',
#   'title': 'html2fo 0.3 (Default)'}, ...
```

The SimpleXMLRPCServer module makes it easy to create straightforward XML-RPC servers. See http://xmlrpc.scripting.com/ for more information about XML-RPC.

- The new hmac module implements the HMAC algorithm described by RFC 2104. (Contributed by Gerhard Häring.)
Several functions that originally returned lengthy tuples now return pseudo-sequences that still behave like tuples but also have mnemonic attributes such as memberst_mtime or tm_year. The enhanced functions include `stat()`, `fstat()`, `statvfs()`, and `fstatvfs()` in the `os` module, and `localtime()`, `gmtime()`, and `strptime()` in the `time` module.

For example, to obtain a file’s size using the old tuples, you’d end up writing something like `file_size = os.stat(filename)[stat.ST_SIZE]`, but now this can be written more clearly as `file_size = os.stat(filename).st_size`.

The original patch for this feature was contributed by Nick Mathewson.

- The Python profiler has been extensively reworked and various errors in its output have been corrected. (Contributed by Fred L. Drake, Jr. and Tim Peters.)

- The `socket` module can be compiled to support IPv6; specify the `--enable-ipv6` option to Python’s configure script. (Contributed by Jun-ichiro “itojun” Hagino.)

- Two new format characters were added to the `struct` module for 64-bit integers on platforms that support the C `long long` type. `q` is for a signed 64-bit integer, and `Q` is for an unsigned one. The value is returned in Python’s long integer type. (Contributed by Tim Peters.)

- In the interpreter’s interactive mode, there’s a new built-in function `help()` that uses the `pydoc` module introduced in Python 2.1 to provide interactive help. `help(object)` displays any available
help text about object. help() with no argument puts you in an online help utility, where you can enter the names of functions, classes, or modules to read their help text. (Contributed by Guido van Rossum, using Ka-Ping Yee’s pydoc module.)

- Various bug fixes and performance improvements have been made to the SRE engine underlying the re module. For example, the re.sub() and re.split() functions have been rewritten in C. Another contributed patch speeds up certain Unicode character ranges by a factor of two, and a new finditer() method that returns an iterator over all the non-overlapping matches in a given string. (SRE is maintained by Fredrik Lundh. The BIGCHARSET patch was contributed by Martin von Löwis.)

- The smtplib module now supports RFC 2487, “Secure SMTP over TLS”, so it’s now possible to encrypt the SMTP traffic between a Python program and the mail transport agent being handed a message. smtplib also supports SMTP authentication. (Contributed by Gerhard Häring.)

- The imaplib module, maintained by Piers Lauder, has support for several new extensions: the NAMESPACE extension defined in RFC 2342, SORT, GETACL and SETACL. (Contributed by Anthony Baxter and Michel Pelletier.)

- The rfc822 module's parsing of email addresses is now compliant with RFC 2822, an update to RFC 822. (The module's name is not going to be changed to rfc2822.) A new package, email, has also been added for parsing and generating e-mail messages. (Contributed by Barry Warsaw, and arising out of his work on Mailman.)
- The `difflib` module now contains a new Differ class for producing human-readable lists of changes (a “delta”) between two sequences of lines of text. There are also two generator functions, ndiff() and restore(), which respectively return a delta from two sequences, or one of the original sequences from a delta. (Grunt work contributed by David Goodger, from ndiff.py code by Tim Peters who then did the generatorization.)

- New constants `ascii_letters`, `ascii_lowercase`, and `ascii_uppercase` were added to the `string` module. There were several modules in the standard library that used `string.letters` to mean the ranges A-Za-z, but that assumption is incorrect when locales are in use, because `string.letters` varies depending on the set of legal characters defined by the current locale. The buggy modules have all been fixed to use `ascii_letters` instead. (Reported by an unknown person; fixed by Fred L. Drake, Jr.)

- The `mimetypes` module now makes it easier to use alternative MIME-type databases by the addition of a `MimeTypes` class, which takes a list of filenames to be parsed. (Contributed by Fred L. Drake, Jr.)

- A Timer class was added to the `threading` module that allows scheduling an activity to happen at some future time. (Contributed by Itamar Shtull-Trauring.)
Interpreter Changes and Fixes

Some of the changes only affect people who deal with the Python interpreter at the C level because they're writing Python extension modules, embedding the interpreter, or just hacking on the interpreter itself. If you only write Python code, none of the changes described here will affect you very much.

- Profiling and tracing functions can now be implemented in C, which can operate at much higher speeds than Python-based functions and should reduce the overhead of profiling and tracing. This will be of interest to authors of development environments for Python. Two new C functions were added to Python's API, PyEval_SetProfile() and PyEval_SetTrace(). The existing sys.setprofile() and sys.settrace() functions still exist, and have simply been changed to use the new C-level interface. (Contributed by Fred L. Drake, Jr.)

- Another low-level API, primarily of interest to implementors of Python debuggers and development tools, was added. PyInterpreterState_Head() and PyInterpreterState_Next() let a caller walk through all the existing interpreter objects; PyInterpreterState_ThreadHead() and PyThreadState_Next() allow looping over all the thread states for a given interpreter. (Contributed by David Beazley.)

- The C-level interface to the garbage collector has been changed to make it easier to write extension types that support garbage collection and to debug misuses of the functions. Various functions have slightly different semantics, so a bunch of functions
had to be renamed. Extensions that use the old API will still compile but will not participate in garbage collection, so updating them for 2.2 should be considered fairly high priority.

To upgrade an extension module to the new API, perform the following steps:

- Rename Py_TPFLAGS_GC() to PyTPFLAGS_HAVE_GC().
- Use PyObject_GC_New() or PyObject_GC_NewVar() to allocate objects, and PyObject_GC_Del() to deallocate them.
- Rename PyObject_GC_Init() to PyObject_GC_Track() and PyObject_GC_Fini() to PyObject_GC_UnTrack().
- Remove PyGC_HEAD_SIZE() from object size calculations.
- Remove calls to PyObject_AS_GC() and PyObject_FROM_GC().
- A new et format sequence was added to PyArg_ParseTuple(); et takes both a parameter and an encoding name, and converts the parameter to the given encoding if the parameter turns out to be a Unicode string, or leaves it alone if it’s an 8-bit string, assuming it to already be in the desired encoding. This differs from the es format character, which assumes that 8-bit strings are in Python’s default ASCII encoding and converts them to the specified new encoding. (Contributed by M.-A. Lemburg, and used for the MBCS support on Windows described in the following section.)
• A different argument parsing function, `PyArg_UnpackTuple()`, has been added that's simpler and presumably faster. Instead of specifying a format string, the caller simply gives the minimum and maximum number of arguments expected, and a set of pointers to `PyObject*` variables that will be filled in with argument values.

• Two new flags `METH_NOARGS` and `METH_0` are available in method definition tables to simplify implementation of methods with no arguments or a single untyped argument. Calling such methods is more efficient than calling a corresponding method that uses `METH_VARARGS`. Also, the old `METH_OLDARGS` style of writing C methods is now officially deprecated.

• Two new wrapper functions, `PyOS_snprintf()` and `PyOS_vsnprintf()` were added to provide cross-platform implementations for the relatively new `snprintf()` and `vsnprintf()` C lib APIs. In contrast to the standard `sprintf()` and `vsprintf()` functions, the Python versions check the bounds of the buffer used to protect against buffer overruns. (Contributed by M.-A. Lemburg.)

• The `_PyTuple_Resize()` function has lost an unused parameter, so now it takes 2 parameters instead of 3. The third argument was never used, and can simply be discarded when porting code from earlier versions to Python 2.2.
Other Changes and Fixes

As usual there were a bunch of other improvements and bug fixes scattered throughout the source tree. A search through the CVS change logs finds there were 527 patches applied and 683 bugs fixed between Python 2.1 and 2.2; 2.2.1 applied 139 patches and fixed 143 bugs; 2.2.2 applied 106 patches and fixed 82 bugs. These figures are likely to be underestimates.

Some of the more notable changes are:

- The code for the MacOS port for Python, maintained by Jack Jansen, is now kept in the main Python CVS tree, and many changes have been made to support MacOS X.

  The most significant change is the ability to build Python as a framework, enabled by supplying the `--enable-framework` option to the configure script when compiling Python. According to Jack Jansen, “This installs a self-contained Python installation plus the OS X framework “glue” into `/Library/Frameworks/Python.framework` (or another location of choice). For now there is little immediate added benefit to this (actually, there is the disadvantage that you have to change your PATH to be able to find Python), but it is the basis for creating a full-blown Python application, porting the MacPython IDE, possibly using Python as a standard OSA scripting language and much more.”

  Most of the MacPython toolbox modules, which interface to MacOS APIs such as windowing, QuickTime, scripting, etc. have been ported to OS X, but they’ve been left commented out in `setup.py`. People who want to experiment with these modules
can uncomment them manually.

- Keyword arguments passed to built-in functions that don’t take them now cause a **TypeError** exception to be raised, with the message “*function* takes no keyword arguments”.

- Weak references, added in Python 2.1 as an extension module, are now part of the core because they’re used in the implementation of new-style classes. The **ReferenceError** exception has therefore moved from the **weakref** module to become a built-in exception.

- A new script, `Tools/scripts/cleanfuture.py` by Tim Peters, automatically removes obsolete **__future__** statements from Python source code.

- An additional **flags** argument has been added to the built-in function **compile()**, so the behaviour of **__future__** statements can now be correctly observed in simulated shells, such as those presented by IDLE and other development environments. This is described in **PEP 264**. (Contributed by Michael Hudson.)

- The new license introduced with Python 1.6 wasn’t GPL-compatible. This is fixed by some minor textual changes to the 2.2 license, so it’s now legal to embed Python inside a GPLed program again. Note that Python itself is not GPLv3ed, but instead is under a license that’s essentially equivalent to the BSD license, same as it always was. The license changes were also applied to the Python 2.0.1 and 2.1.1 releases.

- When presented with a Unicode filename on Windows, Python will now convert it to an MBCS encoded string, as used by the
Microsoft file APIs. As MBCS is explicitly used by the file APIs, Python’s choice of ASCII as the default encoding turns out to be an annoyance. On Unix, the locale’s character set is used if
\texttt{locale.nl\_langinfo(CODESET)} is available. (Windows support was contributed by Mark Hammond with assistance from Marc-André Lemburg. Unix support was added by Martin von Löwis.)

- Large file support is now enabled on Windows. (Contributed by Tim Peters.)

- The \texttt{Tools/scripts/ftpmirror.py} script now parses a \texttt{.netrc} file, if you have one. (Contributed by Mike Romberg.)

- Some features of the object returned by the \texttt{xrange()} function are now deprecated, and trigger warnings when they’re accessed; they’ll disappear in Python 2.3. \texttt{xrange} objects tried to pretend they were full sequence types by supporting slicing, sequence multiplication, and the \texttt{in} operator, but these features were rarely used and therefore buggy. The \texttt{tolist()} method and the \texttt{start}, \texttt{stop}, and \texttt{step} attributes are also being deprecated. At the C level, the fourth argument to the \texttt{PyRange\_New()} function, \texttt{repeat}, has also been deprecated.

- There were a bunch of patches to the dictionary implementation, mostly to fix potential core dumps if a dictionary contains objects that sneakily changed their hash value, or mutated the dictionary they were contained in. For a while python-dev fell into a gentle rhythm of Michael Hudson finding a case that dumped core, Tim Peters fixing the bug, Michael finding another case, and round and round it went.
On Windows, Python can now be compiled with Borland C thanks to a number of patches contributed by Stephen Hansen, though the result isn't fully functional yet. (But this is progress…)

Another Windows enhancement: Wise Solutions generously offered PythonLabs use of their InstallerMaster 8.1 system. Earlier PythonLabs Windows installers used Wise 5.0a, which was beginning to show its age. (Packaged up by Tim Peters.)

Files ending in .pyw can now be imported on Windows. .pyw is a Windows-only thing, used to indicate that a script needs to be run using PYTHONW.EXE instead of PYTHON.EXE in order to prevent a DOS console from popping up to display the output. This patch makes it possible to import such scripts, in case they're also usable as modules. (Implemented by David Bolen.)

On platforms where Python uses the C dlopen() function to load extension modules, it's now possible to set the flags used by dlopen() using the sys.getdlopenflags() and sys.setdlopenflags() functions. (Contributed by Bram Stolk.)

The pow() built-in function no longer supports 3 arguments when floating-point numbers are supplied. pow(x, y, z) returns (x**y) % z, but this is never useful for floating point numbers, and the final result varies unpredictably depending on the platform. A call such as pow(2.0, 8.0, 7.0) will now raise a TypeError exception.
Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Fred Bremmer, Keith Briggs, Andrew Dalke, Fred L. Drake, Jr., Carel Fellinger, David Goodger, Mark Hammond, Stephen Hansen, Michael Hudson, Jack Jansen, Marc-André Lemburg, Martin von Löwis, Fredrik Lundh, Michael McLay, Nick Mathewson, Paul Moore, Gustavo Niemeyer, Don O’Donnell, Joonas Paalasma, Tim Peters, Jens Quade, Tom Reinhardt, Neil Schemenauer, Guido van Rossum, Greg Ward, Edward Welbourne.
What’s New in Python 2.1

Author: A.M. Kuchling
Introduction

This article explains the new features in Python 2.1. While there aren’t as many changes in 2.1 as there were in Python 2.0, there are still some pleasant surprises in store. 2.1 is the first release to be steered through the use of Python Enhancement Proposals, or PEPs, so most of the sizable changes have accompanying PEPs that provide more complete documentation and a design rationale for the change. This article doesn’t attempt to document the new features completely, but simply provides an overview of the new features for Python programmers. Refer to the Python 2.1 documentation, or to the specific PEP, for more details about any new feature that particularly interests you.

One recent goal of the Python development team has been to accelerate the pace of new releases, with a new release coming every 6 to 9 months. 2.1 is the first release to come out at this faster pace, with the first alpha appearing in January, 3 months after the final version of 2.0 was released.

The final release of Python 2.1 was made on April 17, 2001.
PEP 227: Nested Scopes

The largest change in Python 2.1 is to Python’s scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn’t match their intuitive expectations. For example, a nested recursive function definition doesn’t work:

```python
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function `g()` will always raise a `NameError` exception, because the binding of the name `g` isn’t in either its local namespace or in the module-level namespace. This isn’t much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` expression clumsier, and this was a problem in practice. In code which uses `lambda` you can often find local variables being copied by passing them as the default values of arguments.

```python
def find(self, name):
    "Return list of any entries equal to 'name'"
    L = filter(lambda x, name=name: x == name, self.list_attribute)
    return L
```

The readability of Python code written in a strongly functional style
suffers greatly as a result.

The most significant change to Python 2.1 is that static scoping has been added to the language to fix this problem. As a first effect, the name=name default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the def, class, or import statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the from module import * and exec statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that from module import * is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. from module import * and exec make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or lambda expressions with free variables, the compiler will flag this by raising a SyntaxError exception.
To make the preceding explanation a bit clearer, here’s an example:

```python
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
def g():
    return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by `g()`.

This shouldn’t be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it's often a sign of a poor design anyway).

Compatibility concerns have led to nested scopes being introduced gradually; in Python 2.1, they aren’t enabled by default, but can be turned on within a module by using a future statement as described in PEP 236. (See the following section for further discussion of PEP 236.) In Python 2.2, nested scopes will become the default and there will be no way to turn them off, but users will have had all of 2.1’s lifetime to fix any breakage resulting from their introduction.

**See also:**

**PEP 227 - Statically Nested Scopes**

Written and implemented by Jeremy Hylton.
PEP 236: __future__ Directives

The reaction to nested scopes was widespread concern about the dangers of breaking code with the 2.1 release, and it was strong enough to make the Pythoneers take a more conservative approach. This approach consists of introducing a convention for enabling optional functionality in release N that will become compulsory in release N+1.

The syntax uses a `from __future__ import` statement using the reserved module name `__future__`. Nested scopes can be enabled by the following statement:

```
from __future__ import nested_scopes
```

While it looks like a normal `import` statement, it’s not; there are strict rules on where such a future statement can be put. They can only be at the top of a module, and must precede any Python code or regular `import` statements. This is because such statements can affect how the Python bytecode compiler parses code and generates bytecode, so they must precede any statement that will result in bytecodes being produced.

See also:

**PEP 236 - Back to the __future__**

Written by Tim Peters, and primarily implemented by Jeremy Hylton.
PEP 207: Rich Comparisons

In earlier versions, Python's support for implementing comparisons on user-defined classes and extension types was quite simple. Classes could implement a `__cmp__()` method that was given two instances of a class, and could only return 0 if they were equal or +1 or -1 if they weren't; the method couldn't raise an exception or return anything other than a Boolean value. Users of Numeric Python often found this model too weak and restrictive, because in the number-crunching programs that numeric Python is used for, it would be more useful to be able to perform elementwise comparisons of two matrices, returning a matrix containing the results of a given comparison for each element. If the two matrices are of different sizes, then the compare has to be able to raise an exception to signal the error.

In Python 2.1, rich comparisons were added in order to support this need. Python classes can now individually overload each of the `<`, `<=`, `>`, `>=`, `==`, and `!=` operations. The new magic method names are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Method name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>__lt__()</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>__le__()</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>__gt__()</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>__ge__()</code></td>
</tr>
<tr>
<td><code>==</code></td>
<td><code>__eq__()</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td><code>__ne__()</code></td>
</tr>
</tbody>
</table>

(The magic methods are named after the corresponding Fortran operators `.LT.` `.LE.`, &c. Numeric programmers are almost certainly
quite familiar with these names and will find them easy to remember.)

Each of these magic methods is of the form method(self, other), where self will be the object on the left-hand side of the operator, while other will be the object on the right-hand side. For example, the expression A < B will cause A.__lt__(B) to be called.

Each of these magic methods can return anything at all: a Boolean, a matrix, a list, or any other Python object. Alternatively they can raise an exception if the comparison is impossible, inconsistent, or otherwise meaningless.

The built-in cmp(A, B) function can use the rich comparison machinery, and now accepts an optional argument specifying which comparison operation to use; this is given as one of the strings "<", "<=", ">", ">=", "==", or "!=". If called without the optional third argument, cmp() will only return -1, 0, or +1 as in previous versions of Python; otherwise it will call the appropriate method and can return any Python object.

There are also corresponding changes of interest to C programmers; there's a new slot tp_richcmp in type objects and an API for performing a given rich comparison. I won't cover the C API here, but will refer you to PEP 207, or to 2.1’s C API documentation, for the full list of related functions.

See also:

**PEP 207 - Rich Comparisons**

Written by Guido van Rossum, heavily based on earlier work by David Ascher, and implemented by Guido van Rossum.
PEP 230: Warning Framework

Over its 10 years of existence, Python has accumulated a certain number of obsolete modules and features along the way. It's difficult to know when a feature is safe to remove, since there's no way of knowing how much code uses it — perhaps no programs depend on the feature, or perhaps many do. To enable removing old features in a more structured way, a warning framework was added. When the Python developers want to get rid of a feature, it will first trigger a warning in the next version of Python. The following Python version can then drop the feature, and users will have had a full release cycle to remove uses of the old feature.

Python 2.1 adds the warning framework to be used in this scheme. It adds a `warnings` module that provide functions to issue warnings, and to filter out warnings that you don’t want to be displayed. Third-party modules can also use this framework to deprecate old features that they no longer wish to support.

For example, in Python 2.1 the regex module is deprecated, so importing it causes a warning to be printed:

```python
>>> import regex
__main__:1: DeprecationWarning: the regex module is deprecated; please use the re module
```

Warnings can be issued by calling the `warnings.warn()` function:

```python
warnings.warn("feature X no longer supported")
```

The first parameter is the warning message; an additional optional
parameters can be used to specify a particular warning category.

Filters can be added to disable certain warnings; a regular expression pattern can be applied to the message or to the module name in order to suppress a warning. For example, you may have a program that uses the regex module and not want to spare the time to convert it to use the `re` module right now. The warning can be suppressed by calling

```python
import warnings
warnings.filterwarnings(action = 'ignore',
                        message='.*regex module is deprecated',
                        category=DeprecationWarning,
                        module = '__main__')
```

This adds a filter that will apply only to warnings of the class `DeprecationWarning` triggered in the `__main__` module, and applies a regular expression to only match the message about the regex module being deprecated, and will cause such warnings to be ignored. Warnings can also be printed only once, printed every time the offending code is executed, or turned into exceptions that will cause the program to stop (unless the exceptions are caught in the usual way, of course).

Functions were also added to Python’s C API for issuing warnings; refer to PEP 230 or to Python’s API documentation for the details.

**See also:**

**PEP 5 - Guidelines for Language Evolution**

Written by Paul Prescod, to specify procedures to be followed when removing old features from Python. The policy described in this PEP hasn’t been officially adopted, but the eventual policy
probably won’t be too different from Prescod’s proposal.

**PEP 230 - Warning Framework**
Written and implemented by Guido van Rossum.
PEP 229: New Build System

When compiling Python, the user had to go in and edit the Modules/Setup file in order to enable various additional modules; the default set is relatively small and limited to modules that compile on most Unix platforms. This means that on Unix platforms with many more features, most notably Linux, Python installations often don’t contain all useful modules they could.

Python 2.0 added the Distutils, a set of modules for distributing and installing extensions. In Python 2.1, the Distutils are used to compile much of the standard library of extension modules, autodetecting which ones are supported on the current machine. It’s hoped that this will make Python installations easier and more featureful.

Instead of having to edit the Modules/Setup file in order to enable modules, a setup.py script in the top directory of the Python source distribution is run at build time, and attempts to discover which modules can be enabled by examining the modules and header files on the system. If a module is configured in Modules/Setup, the setup.py script won’t attempt to compile that module and will defer to the Modules/Setup file’s contents. This provides a way to specific any strange command-line flags or libraries that are required for a specific platform.

In another far-reaching change to the build mechanism, Neil Schemenauer restructured things so Python now uses a single makefile that isn’t recursive, instead of makefiles in the top directory and in each of the Python/, Parser/, Objects/, and Modules/ subdirectories. This makes building Python faster and also makes hacking the Makefiles clearer and simpler.
See also:

**PEP 229** - Using Distutils to Build Python
Written and implemented by A.M. Kuchling.
PEP 205: Weak References

Weak references, available through the `weakref` module, are a minor but useful new data type in the Python programmer’s toolbox.

Storing a reference to an object (say, in a dictionary or a list) has the side effect of keeping that object alive forever. There are a few specific cases where this behaviour is undesirable, object caches being the most common one, and another being circular references in data structures such as trees.

For example, consider a memoizing function that caches the results of another function $f(x)$ by storing the function’s argument and its result in a dictionary:

```python
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        return _cache[x]

    retval = f(x)

    # Cache the returned object
    _cache[x] = retval

    return retval
```

This version works for simple things such as integers, but it has a side effect; the `_cache` dictionary holds a reference to the return values, so they’ll never be deallocated until the Python process exits and cleans up. This isn’t very noticeable for integers, but if $f()$ returns an object, or a data structure that takes up a lot of memory, this can be a problem.
Weak references provide a way to implement a cache that won’t keep objects alive beyond their time. If an object is only accessible through weak references, the object will be deallocated and the weak references will now indicate that the object it referred to no longer exists. A weak reference to an object \texttt{obj} is created by calling \texttt{wr = weakref.ref(obj)}. The object being referred to is returned by calling the weak reference as if it were a function: \texttt{wr()}. It will return the referenced object, or \texttt{None} if the object no longer exists.

This makes it possible to write a \texttt{memoize()} function whose cache doesn’t keep objects alive, by storing weak references in the cache.

```python
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        obj = _cache[x]()
        # If \texttt{weak reference object} still exists,
        # return it
        if obj is not None: return obj

    retval = f(x)

    # Cache a \texttt{weak reference}
    _cache[x] = weakref.ref(retval)

    return retval
```

The \texttt{weakref} module also allows creating proxy objects which behave like weak references — an object referenced only by proxy objects is deallocated – but instead of requiring an explicit call to retrieve the object, the proxy transparently forwards all operations to the object as long as the object still exists. If the object is deallocated, attempting to use a proxy will cause a `weakref.ReferenceError` exception to be raised.
```python
proxy = weakref.proxy(obj)
proxy.attr       # Equivalent to obj.attr
proxy.meth()    # Equivalent to obj.meth()
    del obj
proxy.attr       # raises weakref.ReferenceError
```

See also:

**PEP 205 - Weak References**

Written and implemented by Fred L. Drake, Jr.
In Python 2.1, functions can now have arbitrary information attached to them. People were often using docstrings to hold information about functions and methods, because the __doc__ attribute was the only way of attaching any information to a function. For example, in the Zope Web application server, functions are marked as safe for public access by having a docstring, and in John Aycock’s SPARK parsing framework, docstrings hold parts of the BNF grammar to be parsed. This overloading is unfortunate, since docstrings are really intended to hold a function's documentation; for example, it means you can't properly document functions intended for private use in Zope.

Arbitrary attributes can now be set and retrieved on functions using the regular Python syntax:

```python
def f(): pass
f.publish = 1
f.secure = 1
f.grammar = "A ::= B (C D)*"
```

The dictionary containing attributes can be accessed as the function's __dict__. Unlike the __dict__ attribute of class instances, in functions you can actually assign a new dictionary to __dict__, though the new value is restricted to a regular Python dictionary; you can't be tricky and set it to a UserDict instance, or any other random object that behaves like a mapping.
Written and implemented by Barry Warsaw.
PEP 235: Importing Modules on Case-Insensitive Platforms

Some operating systems have filesystems that are case-insensitive, MacOS and Windows being the primary examples; on these systems, it's impossible to distinguish the filenames FILE.PY and file.py, even though they do store the file's name in its original case (they're case-preserving, too).

In Python 2.1, the `import` statement will work to simulate case-sensitivity on case-insensitive platforms. Python will now search for the first case-sensitive match by default, raising an `ImportError` if no such file is found, so `import file` will not import a module named FILE.PY. Case-insensitive matching can be requested by setting the `PYTHONCASEOK` environment variable before starting the Python interpreter.
PEP 217: Interactive Display Hook

When using the Python interpreter interactively, the output of commands is displayed using the built-in `repr()` function. In Python 2.1, the variable `sys.displayhook()` can be set to a callable object which will be called instead of `repr()`. For example, you can set it to a special pretty-printing function:

```python
>>> # Create a recursive data structure
... L = [1,2,3]
>>> L.append(L)
>>> L # Show Python's default output
[1, 2, 3, [...]]
>>> # Use pprint.pprint() as the display function
... import sys, pprint
>>> sys.displayhook = pprint.pprint
>>> L
[1, 2, 3,  <Recursion on list with id=135143996>]
```
PEP 208: New Coercion Model

How numeric coercion is done at the C level was significantly modified. This will only affect the authors of C extensions to Python, allowing them more flexibility in writing extension types that support numeric operations.

Extension types can now set the type flag Py_TPFLAGS_CHECKTYPES in their PyTypeObject structure to indicate that they support the new coercion model. In such extension types, the numeric slot functions can no longer assume that they’ll be passed two arguments of the same type; instead they may be passed two arguments of differing types, and can then perform their own internal coercion. If the slot function is passed a type it can’t handle, it can indicate the failure by returning a reference to the Py_NotImplemented singleton value. The numeric functions of the other type will then be tried, and perhaps they can handle the operation; if the other type also returns Py_NotImplemented, then a TypeError will be raised. Numeric methods written in Python can also return Py_NotImplemented, causing the interpreter to act as if the method did not exist (perhaps raising a TypeError, perhaps trying another object’s numeric methods).

See also:

PEP 208 - Reworking the Coercion Model
Written and implemented by Neil Schemenauer, heavily based upon earlier work by Marc-André Lemburg. Read this to understand the fine points of how numeric operations will now be processed at the C level.
PEP 241: Metadata in Python Packages

A common complaint from Python users is that there's no single catalog of all the Python modules in existence. T. Middleton's Vaults of Parnassus at http://www.vex.net/parnassus/ are the largest catalog of Python modules, but registering software at the Vaults is optional, and many people don't bother.

As a first small step toward fixing the problem, Python software packaged using the Distutils sdist command will include a file named PKG-INFO containing information about the package such as its name, version, and author (metadata, in cataloguing terminology). PEP 241 contains the full list of fields that can be present in the PKG-INFO file. As people began to package their software using Python 2.1, more and more packages will include metadata, making it possible to build automated cataloguing systems and experiment with them. With the result experience, perhaps it'll be possible to design a really good catalog and then build support for it into Python 2.2. For example, the Distutils sdist and bdist_* commands could support an upload option that would automatically upload your package to a catalog server.

You can start creating packages containing PKG-INFO even if you're not using Python 2.1, since a new release of the Distutils will be made for users of earlier Python versions. Version 1.0.2 of the Distutils includes the changes described in PEP 241, as well as various bugfixes and enhancements. It will be available from the Distutils SIG at https://www.python.org/community/sigs/current/distutils-sig/.

See also:
**PEP 241 - Metadata for Python Software Packages**  
Written and implemented by A.M. Kuchling.

**PEP 243 - Module Repository Upload Mechanism**  
Written by Sean Reifschneider, this draft PEP describes a proposed mechanism for uploading Python packages to a central server.
New and Improved Modules

- Ka-Ping Yee contributed two new modules: inspect.py, a module for getting information about live Python code, and pydoc.py, a module for interactively converting docstrings to HTML or text. As a bonus, Tools/scripts/pydoc, which is now automatically installed, uses pydoc.py to display documentation given a Python module, package, or class name. For example, `pydoc xml.dom` displays the following:

```
Python Library Documentation: package xml.dom in

NAME
xml.dom - W3C Document Object Model implementation

FILE
/usr/local/lib/python2.1/xml/dom/__init__.pyc

DESCRIPTION
The Python mapping of the Document Object Model

Python Library Reference in the section on the

This package contains the following modules:
...```

pydoc also includes a Tk-based interactive help browser. pydoc quickly becomes addictive; try it out!

- Two different modules for unit testing were added to the standard library. The doctest module, contributed by Tim Peters, provides a testing framework based on running embedded examples in docstrings and comparing the results against the expected output.
PyUnit, contributed by Steve Purcell, is a unit testing framework inspired by JUnit, which was in turn an adaptation of Kent Beck's Smalltalk testing framework. See http://pyunit.sourceforge.net/ for more information about PyUnit.

- The **difflib** module contains a class, SequenceMatcher, which compares two sequences and computes the changes required to transform one sequence into the other. For example, this module can be used to write a tool similar to the Unix **diff** program, and in fact the sample program `Tools/scripts/ndiff.py` demonstrates how to write such a script.

- **curses.panel**, a wrapper for the panel library, part of ncurses and of SYSV curses, was contributed by Thomas Gellekum. The panel library provides windows with the additional feature of depth. Windows can be moved higher or lower in the depth ordering, and the panel library figures out where panels overlap and which sections are visible.

- The PyXML package has gone through a few releases since Python 2.0, and Python 2.1 includes an updated version of the xml package. Some of the noteworthy changes include support for Expat 1.2 and later versions, the ability for Expat parsers to handle files in any encoding supported by Python, and various bugfixes for SAX, DOM, and the minidom module.

- Ping also contributed another hook for handling uncaught exceptions. `sys.excepthook()` can be set to a callable object. When an exception isn’t caught by any `try...except` blocks, the exception will be passed to `sys.excepthook()`, which can then do whatever it likes. At the Ninth Python Conference, Ping
demonstrated an application for this hook: printing an extended traceback that not only lists the stack frames, but also lists the function arguments and the local variables for each frame.

- Various functions in the `time` module, such as `asctime()` and `localtime()`, require a floating point argument containing the time in seconds since the epoch. The most common use of these functions is to work with the current time, so the floating point argument has been made optional; when a value isn’t provided, the current time will be used. For example, log file entries usually need a string containing the current time; in Python 2.1, `time.asctime()` can be used, instead of the lengthier `time.asctime(time.localtime(time.time()))` that was previously required.

  This change was proposed and implemented by Thomas Wouters.

- The `ftplib` module now defaults to retrieving files in passive mode, because passive mode is more likely to work from behind a firewall. This request came from the Debian bug tracking system, since other Debian packages use `ftplib` to retrieve files and then don’t work from behind a firewall. It’s deemed unlikely that this will cause problems for anyone, because Netscape defaults to passive mode and few people complain, but if passive mode is unsuitable for your application or network setup, call `set_pasv(0)` on FTP objects to disable passive mode.

- Support for raw socket access has been added to the `socket` module, contributed by Grant Edwards.

- The `pstats` module now contains a simple interactive statistics
browser for displaying timing profiles for Python programs, invoked when the module is run as a script. Contributed by Eric S. Raymond.

- A new implementation-dependent function, `sys._getframe([depth])`, has been added to return a given frame object from the current call stack. `sys._getframe()` returns the frame at the top of the call stack; if the optional integer argument `depth` is supplied, the function returns the frame that is `depth` calls below the top of the stack. For example, `sys._getframe(1)` returns the caller’s frame object.

This function is only present in CPython, not in Jython or the .NET implementation. Use it for debugging, and resist the temptation to put it into production code.
Other Changes and Fixes

There were relatively few smaller changes made in Python 2.1 due to the shorter release cycle. A search through the CVS change logs turns up 117 patches applied, and 136 bugs fixed; both figures are likely to be underestimates. Some of the more notable changes are:

- A specialized object allocator is now optionally available, that should be faster than the system malloc() and have less memory overhead. The allocator uses C's malloc() function to get large pools of memory, and then fulfills smaller memory requests from these pools. It can be enabled by providing the --with-pymalloc option to the configure script; see Objects/obmalloc.c for the implementation details.

Authors of C extension modules should test their code with the object allocator enabled, because some incorrect code may break, causing core dumps at runtime. There are a bunch of memory allocation functions in Python's C API that have previously been just aliases for the C library's malloc() and free(), meaning that if you accidentally called mismatched functions, the error wouldn't be noticeable. When the object allocator is enabled, these functions aren't aliases of malloc() and free() any more, and calling the wrong function to free memory will get you a core dump. For example, if memory was allocated using PyMem_New(), it has to be freed using PyMem_Del(), not free(). A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will have the same problem.

The object allocator was contributed by Vladimir Marangozov.
The speed of line-oriented file I/O has been improved because people often complain about its lack of speed, and because it's often been used as a naïve benchmark. The `readline()` method of file objects has therefore been rewritten to be much faster. The exact amount of the speedup will vary from platform to platform depending on how slow the C library's `getc()` was, but is around 66%, and potentially much faster on some particular operating systems. Tim Peters did much of the benchmarking and coding for this change, motivated by a discussion in comp.lang.python.

A new module and method for file objects was also added, contributed by Jeff Epler. The new method, `xreadlines()`, is similar to the existing `xrange()` built-in. `xreadlines()` returns an opaque sequence object that only supports being iterated over, reading a line on every iteration but not reading the entire file into memory as the existing `readlines()` method does. You'd use it like this:

```python
for line in sys.stdin.xreadlines():
    # ... do something for each line ...
...
```

For a fuller discussion of the line I/O changes, see the python-dev summary for January 1–15, 2001 at https://mail.python.org/pipermail/python-dev/2001-January/.

A new method, `popitem()`, was added to dictionaries to enable destructively iterating through the contents of a dictionary; this can be faster for large dictionaries because there's no need to construct a list containing all the keys or values. `D.popitem()` removes a random (key, value) pair from the dictionary `D` and
returns it as a 2-tuple. This was implemented mostly by Tim Peters and Guido van Rossum, after a suggestion and preliminary patch by Moshe Zadka.

- Modules can now control which names are imported when `from module import *` is used, by defining an `__all__` attribute containing a list of names that will be imported. One common complaint is that if the module imports other modules such as `sys` or `string`, `from module import *` will add them to the importing module's namespace. To fix this, simply list the public names in `__all__`:

  ```
  # List public names
  __all__ = ['Database', 'open']
  ```

  A stricter version of this patch was first suggested and implemented by Ben Wolfson, but after some python-dev discussion, a weaker final version was checked in.

- Applying `repr()` to strings previously used octal escapes for non-printable characters; for example, a newline was '\012'. This was a vestigial trace of Python’s C ancestry, but today octal is of very little practical use. Ka-Ping Yee suggested using hex escapes instead of octal ones, and using the `\n, \t, \r` escapes for the appropriate characters, and implemented this new formatting.

- Syntax errors detected at compile-time can now raise exceptions containing the filename and line number of the error, a pleasant side effect of the compiler reorganization done by Jeremy Hylton.

- C extensions which import other modules have been changed to use `PyImport_ImportModule()`, which means that they will
use any import hooks that have been installed. This is also encouraged for third-party extensions that need to import some other module from C code.

- The size of the Unicode character database was shrunk by another 340K thanks to Fredrik Lundh.

- Some new ports were contributed: MacOS X (by Steven Majewski), Cygwin (by Jason Tishler); RISCOS (by Dietmar Schwertberger); Unixware 7 (by Billy G. Allie).

And there's the usual list of minor bugfixes, minor memory leaks, docstring edits, and other tweaks, too lengthy to be worth itemizing; see the CVS logs for the full details if you want them.
Acknowledgements

The author would like to thank the following people for offering suggestions on various drafts of this article: Graeme Cross, David Goodger, Jay Graves, Michael Hudson, Marc-André Lemburg, Fredrik Lundh, Neil Schemenauer, Thomas Wouters.
What’s New in Python 2.0

Author: A.M. Kuchling and Moshe Zadka
Introduction

A new release of Python, version 2.0, was released on October 16, 2000. This article covers the exciting new features in 2.0, highlights some other useful changes, and points out a few incompatible changes that may require rewriting code.

Python’s development never completely stops between releases, and a steady flow of bug fixes and improvements are always being submitted. A host of minor fixes, a few optimizations, additional docstrings, and better error messages went into 2.0; to list them all would be impossible, but they’re certainly significant. Consult the publicly-available CVS logs if you want to see the full list. This progress is due to the five developers working for PythonLabs are now getting paid to spend their days fixing bugs, and also due to the improved communication resulting from moving to SourceForge.
What About Python 1.6?

Python 1.6 can be thought of as the Contractual Obligations Python release. After the core development team left CNRI in May 2000, CNRI requested that a 1.6 release be created, containing all the work on Python that had been performed at CNRI. Python 1.6 therefore represents the state of the CVS tree as of May 2000, with the most significant new feature being Unicode support. Development continued after May, of course, so the 1.6 tree received a few fixes to ensure that it's forward-compatible with Python 2.0. 1.6 is therefore part of Python's evolution, and not a side branch.

So, should you take much interest in Python 1.6? Probably not. The 1.6final and 2.0beta1 releases were made on the same day (September 5, 2000), the plan being to finalize Python 2.0 within a month or so. If you have applications to maintain, there seems little point in breaking things by moving to 1.6, fixing them, and then having another round of breakage within a month by moving to 2.0; you're better off just going straight to 2.0. Most of the really interesting features described in this document are only in 2.0, because a lot of work was done between May and September.
New Development Process

The most important change in Python 2.0 may not be to the code at all, but to how Python is developed: in May 2000 the Python developers began using the tools made available by SourceForge for storing source code, tracking bug reports, and managing the queue of patch submissions. To report bugs or submit patches for Python 2.0, use the bug tracking and patch manager tools available from Python's project page, located at https://sourceforge.net/projects/python/.

The most important of the services now hosted at SourceForge is the Python CVS tree, the version-controlled repository containing the source code for Python. Previously, there were roughly 7 or so people who had write access to the CVS tree, and all patches had to be inspected and checked in by one of the people on this short list. Obviously, this wasn’t very scalable. By moving the CVS tree to SourceForge, it became possible to grant write access to more people; as of September 2000 there were 27 people able to check in changes, a fourfold increase. This makes possible large-scale changes that wouldn’t be attempted if they’d have to be filtered through the small group of core developers. For example, one day Peter Schneider-Kamp took it into his head to drop K&R C compatibility and convert the C source for Python to ANSI C. After getting approval on the python-dev mailing list, he launched into a flurry of checkins that lasted about a week, other developers joined in to help, and the job was done. If there were only 5 people with write access, probably that task would have been viewed as “nice, but not worth the time and effort needed” and it would never have gotten done.

The shift to using SourceForge's services has resulted in a remarkable increase in the speed of development. Patches now get submitted,
commented on, revised by people other than the original submitter, and bounced back and forth between people until the patch is deemed worth checking in. Bugs are tracked in one central location and can be assigned to a specific person for fixing, and we can count the number of open bugs to measure progress. This didn’t come without a cost: developers now have more e-mail to deal with, more mailing lists to follow, and special tools had to be written for the new environment. For example, SourceForge sends default patch and bug notification e-mail messages that are completely unhelpful, so Ka-Ping Yee wrote an HTML screen-scraper that sends more useful messages.

The ease of adding code caused a few initial growing pains, such as code was checked in before it was ready or without getting clear agreement from the developer group. The approval process that has emerged is somewhat similar to that used by the Apache group. Developers can vote +1, +0, -0, or -1 on a patch; +1 and -1 denote acceptance or rejection, while +0 and -0 mean the developer is mostly indifferent to the change, though with a slight positive or negative slant. The most significant change from the Apache model is that the voting is essentially advisory, letting Guido van Rossum, who has Benevolent Dictator For Life status, know what the general opinion is. He can still ignore the result of a vote, and approve or reject a change even if the community disagrees with him.

Producing an actual patch is the last step in adding a new feature, and is usually easy compared to the earlier task of coming up with a good design. Discussions of new features can often explode into lengthy mailing list threads, making the discussion hard to follow, and no one can read every posting to python-dev. Therefore, a relatively formal process has been set up to write Python Enhancement Proposals (PEPs), modelled on the Internet RFC process. PEPs are draft documents that describe a proposed new feature, and are continually
revised until the community reaches a consensus, either accepting or rejecting the proposal. Quoting from the introduction to PEP 1, “PEP Purpose and Guidelines”:

PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python. The PEP should provide a concise technical specification of the feature and a rationale for the feature.

We intend PEPs to be the primary mechanisms for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

Read the rest of PEP 1 for the details of the PEP editorial process, style, and format. PEPs are kept in the Python CVS tree on SourceForge, though they’re not part of the Python 2.0 distribution, and are also available in HTML form from https://www.python.org/dev/peps/. As of September 2000, there are 25 PEPs, ranging from PEP 201, “Lockstep Iteration”, to PEP 225, “Elementwise/Objectwise Operators”.
Unicode

The largest new feature in Python 2.0 is a new fundamental data type: Unicode strings. Unicode uses 16-bit numbers to represent characters instead of the 8-bit number used by ASCII, meaning that 65,536 distinct characters can be supported.

The final interface for Unicode support was arrived at through countless often-stormy discussions on the python-dev mailing list, and mostly implemented by Marc-André Lemburg, based on a Unicode string type implementation by Fredrik Lundh. A detailed explanation of the interface was written up as PEP 100, “Python Unicode Integration”. This article will simply cover the most significant points about the Unicode interfaces.

In Python source code, Unicode strings are written as u"string". Arbitrary Unicode characters can be written using a new escape sequence, \uHHHH, where HHHH is a 4-digit hexadecimal number from 0000 to FFFF. The existing \xHHHH escape sequence can also be used, and octal escapes can be used for characters up to U+01FF, which is represented by \777.

Unicode strings, just like regular strings, are an immutable sequence type. They can be indexed and sliced, but not modified in place. Unicode strings have an encode([encoding]) method that returns an 8-bit string in the desired encoding. Encodings are named by strings, such as 'ascii', 'utf-8', 'iso-8859-1', or whatever. A codec API is defined for implementing and registering new encodings that are then available throughout a Python program. If an encoding isn’t specified, the default encoding is usually 7-bit ASCII, though it can be changed for your Python installation by calling the
sys.setdefaultencoding(encoding) function in a customized version of site.py.

Combining 8-bit and Unicode strings always coerces to Unicode, using the default ASCII encoding; the result of 'a' + u'bc' is u'abc'.

New built-in functions have been added, and existing built-ins modified to support Unicode:

- **unichr(ch)** returns a Unicode string 1 character long, containing the character ch.
- **ord(u)**, where u is a 1-character regular or Unicode string, returns the number of the character as an integer.
- **unicode(string [, encoding] [, errors] )** creates a Unicode string from an 8-bit string. encoding is a string naming the encoding to use. The errors parameter specifies the treatment of characters that are invalid for the current encoding; passing 'strict' as the value causes an exception to be raised on any encoding error, while 'ignore' causes errors to be silently ignored and 'replace' uses U+FFFD, the official replacement character, in case of any problems.
- The exec statement, and various built-ins such as eval(), getattr(), and setattr() will also accept Unicode strings as well as regular strings. (It's possible that the process of fixing this missed some built-ins; if you find a built-in function that accepts strings but doesn't accept Unicode strings at all, please report it as a bug.)

A new module, **unicodedata**, provides an interface to Unicode character properties. For example, **unicodedata.category(u'A')** returns the 2-character string ‘Lu’, the ‘L’ denoting it’s a letter, and ‘u’ meaning that it’s uppercase. **unicodedata.bidirectional(u'\u0660')** returns ‘AN’. 
meaning that U+0660 is an Arabic number.

The `codecs` module contains functions to look up existing encodings and register new ones. Unless you want to implement a new encoding, you'll most often use the `codecs.lookup(encoding)` function, which returns a 4-element tuple: `(encode_func, decode_func, stream_reader, stream_writer).

- `encode_func` is a function that takes a Unicode string, and returns a 2-tuple `(string, length)`. `string` is an 8-bit string containing a portion (perhaps all) of the Unicode string converted into the given encoding, and `length` tells you how much of the Unicode string was converted.
- `decode_func` is the opposite of `encode_func`, taking an 8-bit string and returning a 2-tuple `(ustring, length)`, consisting of the resulting Unicode string `ustring` and the integer `length` telling how much of the 8-bit string was consumed.
- `stream_reader` is a class that supports decoding input from a stream. `stream_reader(file_obj)` returns an object that supports the `read()`, `readline()`, and `readlines()` methods. These methods will all translate from the given encoding and return Unicode strings.
- `stream_writer`, similarly, is a class that supports encoding output to a stream. `stream_writer(file_obj)` returns an object that supports the `write()` and `writelines()` methods. These methods expect Unicode strings, translating them to the given encoding on output.

For example, the following code writes a Unicode string into a file, encoding it as UTF-8:

```python
import codecs
unistr = u'\u0660\u2000ab ...'
```
UTF8_encode, UTF8_decode, UTF8_streamreader, UTF8_streamwriter = codecs.lookup

output = UTF8_streamwriter(open('/tmp/output', 'wb'))
output.write(unistr)
output.close()

The following code would then read UTF-8 input from the file:

input = UTF8_streamreader(open('/tmp/output', 'rb'))
print repr(input.read())
input.close()

Unicode-aware regular expressions are available through the re module, which has a new underlying implementation called SRE written by Fredrik Lundh of Secret Labs AB.

A -U command line option was added which causes the Python compiler to interpret all string literals as Unicode string literals. This is intended to be used in testing and future-proofing your Python code, since some future version of Python may drop support for 8-bit strings and provide only Unicode strings.
List Comprehensions

Lists are a workhorse data type in Python, and many programs manipulate a list at some point. Two common operations on lists are to loop over them, and either pick out the elements that meet a certain criterion, or apply some function to each element. For example, given a list of strings, you might want to pull out all the strings containing a given substring, or strip off trailing whitespace from each line.

The existing `map()` and `filter()` functions can be used for this purpose, but they require a function as one of their arguments. This is fine if there's an existing built-in function that can be passed directly, but if there isn't, you have to create a little function to do the required work, and Python's scoping rules make the result ugly if the little function needs additional information. Take the first example in the previous paragraph, finding all the strings in the list containing a given substring. You could write the following to do it:

```python
# Given the list L, make a list of all strings
# containing the substring S.
sublist = filter(lambda s, substring=S:
    string.find(s, substring) != -1,
    L)
```

Because of Python's scoping rules, a default argument is used so that the anonymous function created by the `lambda` expression knows what substring is being searched for. List comprehensions make this cleaner:

```python
sublist = [ s for s in L if string.find(s, S) != -1
```

List comprehensions have the form:

```
[ expression for expr in sequence1
  for expr2 in sequence2 ...
  for exprN in sequenceN
  if condition ]
```

The for...in clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are not iterated over in parallel, but from left to right; this is explained more clearly in the following paragraphs. The elements of the generated list will be the successive values of expression. The final if clause is optional; if present, expression is only evaluated and added to the result if condition is true.

To make the semantics very clear, a list comprehension is equivalent to the following Python code:

```
for expr1 in sequence1:
  for expr2 in sequence2:
    ...
  for exprN in sequenceN:
    if (condition):
      # Append the value of
      # the expression to the
      # resulting list.
```

This means that when there are multiple for...in clauses, the resulting list will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```
seq1 = 'abc'
seq2 = (1,2,3)
```
To avoid introducing an ambiguity into Python's grammar, if expression is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```python
>>> [(x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('c', 2), ('c', 3)]
```

The idea of list comprehensions originally comes from the functional programming language Haskell ([https://www.haskell.org](https://www.haskell.org)). Greg Ewing argued most effectively for adding them to Python and wrote the initial list comprehension patch, which was then discussed for a seemingly endless time on the python-dev mailing list and kept up-to-date by Skip Montanaro.
Augmented Assignment

Augmented assignment operators, another long-requested feature, have been added to Python 2.0. Augmented assignment operators include `+=`, `-=``, `*=``, and so forth. For example, the statement `a += 2` increments the value of the variable `a` by 2, equivalent to the slightly lengthier `a = a + 2`.

The full list of supported assignment operators is `+=`, `-=``, `*=``, `/=``, `%=``, `**=``, `&=``, `|=``, `^=``, `>>=``, and `<<=`. Python classes can override the augmented assignment operators by defining methods named `_iadd__`, `_isub__`, etc. For example, the following `Number` class stores a number and supports using `+=` to create a new instance with an incremented value.

```python
class Number:
    def __init__(self, value):
        self.value = value
    def __iadd__(self, increment):
        return Number( self.value + increment)

n = Number(5)
n += 3
print n.value
```

The `_iadd__` special method is called with the value of the increment, and should return a new instance with an appropriately modified value; this return value is bound as the new value of the variable on the left-hand side.

Augmented assignment operators were first introduced in the C programming language, and most C-derived languages, such as `awk`,
C++, Java, Perl, and PHP also support them. The augmented assignment patch was implemented by Thomas Wouters.
String Methods

Until now string-manipulation functionality was in the \texttt{string} module, which was usually a front-end for the \texttt{strop} module written in C. The addition of Unicode posed a difficulty for the \texttt{strop} module, because the functions would all need to be rewritten in order to accept either 8-bit or Unicode strings. For functions such as \texttt{string.replace()}, which takes 3 string arguments, that means eight possible permutations, and correspondingly complicated code.

Instead, Python 2.0 pushes the problem onto the string type, making string manipulation functionality available through methods on both 8-bit strings and Unicode strings.

\begin{verbatim}
>>> 'andrew'.capitalize()
'Andrew'
>>> 'hostname'.replace('os', 'linux')
'hlinuxtname'
>>> 'moshe'.find('sh')
2
\end{verbatim}

One thing that hasn't changed, a noteworthy April Fools' joke notwithstanding, is that Python strings are immutable. Thus, the string methods return new strings, and do not modify the string on which they operate.

The old \texttt{string} module is still around for backwards compatibility, but it mostly acts as a front-end to the new string methods.

Two methods which have no parallel in pre-2.0 versions, although they did exist in JPython for quite some time, are \texttt{startswith()} and \texttt{endswith()}. \texttt{s.startswith(t)} is equivalent to \texttt{s[:\len(t)] ==

\begin{verbatim}

\end{verbatim}
t, while \texttt{s.endswith(t)} is equivalent to \texttt{s[-len(t):] == t}.

One other method which deserves special mention is \texttt{join()}. The \texttt{join()} method of a string receives one parameter, a sequence of strings, and is equivalent to the \texttt{string.join()} function from the old \texttt{string} module, with the arguments reversed. In other words, \texttt{s.join(seq)} is equivalent to the old \texttt{string.join(seq, s)}. 
Garbage Collection of Cycles

The C implementation of Python uses reference counting to implement garbage collection. Every Python object maintains a count of the number of references pointing to it, and adjusts the count as references are created or destroyed. Once the reference count reaches zero, the object is no longer accessible, since you need to have a reference to an object to access it, and if the count is zero, no references exist any longer.

Reference counting has some pleasant properties: it's easy to understand and implement, and the resulting implementation is portable, fairly fast, and reacts well with other libraries that implement their own memory handling schemes. The major problem with reference counting is that it sometimes doesn't realise that objects are no longer accessible, resulting in a memory leak. This happens when there are cycles of references.

Consider the simplest possible cycle, a class instance which has a reference to itself:

```python
instance = SomeClass()
instance.myself = instance
```

After the above two lines of code have been executed, the reference count of `instance` is 2; one reference is from the variable named 'instance', and the other is from the `myself` attribute of the instance.

If the next line of code is `del instance`, what happens? The reference count of `instance` is decreased by 1, so it has a reference count of 1; the reference in the `myself` attribute still exists. Yet the
instance is no longer accessible through Python code, and it could be deleted. Several objects can participate in a cycle if they have references to each other, causing all of the objects to be leaked.

Python 2.0 fixes this problem by periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. A new gc module provides functions to perform a garbage collection, obtain debugging statistics, and tuning the collector’s parameters.

Running the cycle detection algorithm takes some time, and therefore will result in some additional overhead. It is hoped that after we've gotten experience with the cycle collection from using 2.0, Python 2.1 will be able to minimize the overhead with careful tuning. It's not yet obvious how much performance is lost, because benchmarking this is tricky and depends crucially on how often the program creates and destroys objects. The detection of cycles can be disabled when Python is compiled, if you can’t afford even a tiny speed penalty or suspect that the cycle collection is buggy, by specifying the --without-cycle-gc switch when running the configure script.

Several people tackled this problem and contributed to a solution. An early implementation of the cycle detection approach was written by Toby Kelsey. The current algorithm was suggested by Eric Tiedemann during a visit to CNRI, and Guido van Rossum and Neil Schemenauer wrote two different implementations, which were later integrated by Neil. Lots of other people offered suggestions along the way; the March 2000 archives of the python-dev mailing list contain most of the relevant discussion, especially in the threads titled “Reference cycle collection for Python” and “Finalization again”.
Other Core Changes

Various minor changes have been made to Python’s syntax and built-in functions. None of the changes are very far-reaching, but they’re handy conveniences.

Minor Language Changes

A new syntax makes it more convenient to call a given function with a tuple of arguments and/or a dictionary of keyword arguments. In Python 1.5 and earlier, you’d use the `apply()` built-in function: `apply(f, args, kw)` calls the function `f()` with the argument tuple `args` and the keyword arguments in the dictionary `kw`. `apply()` is the same in 2.0, but thanks to a patch from Greg Ewing, `f(*args, **kw)` is a shorter and clearer way to achieve the same effect. This syntax is symmetrical with the syntax for defining functions:

```python
def f(*args, **kw):
    # args is a tuple of positional args,
    # kw is a dictionary of keyword args
    ...
```

The `print` statement can now have its output directed to a file-like object by following the `print` with `>> file`, similar to the redirection operator in Unix shells. Previously you’d either have to use the `write()` method of the file-like object, which lacks the convenience and simplicity of `print`, or you could assign a new value to `sys.stdout` and then restore the old value. For sending output to standard error, it’s much easier to write this:

```python
print >> sys.stderr, "Warning: action field not supp"
Modules can now be renamed on importing them, using the syntax
\[
\text{import module as name or from module import name as othername.}
\]
The patch was submitted by Thomas Wouters.

A new format style is available when using the % operator; ‘%r’ will
insert the \texttt{repr()} of its argument. This was also added from
symmetry considerations, this time for symmetry with the existing ‘%s’
format style, which inserts the \texttt{str()} of its argument. For example,
\[
\text{’%r %s’ % (’abc’, ’abc’)}
\]
returns a string containing \texttt{’abc’}.

Previously there was no way to implement a class that overrode
Python's built-in \texttt{in} operator and implemented a custom version. \texttt{obj in seq}
returns true if \texttt{obj} is present in the sequence \texttt{seq}; Python
computes this by simply trying every index of the sequence until either
\texttt{obj} is found or an \texttt{IndexError} is encountered. Moshe Zadka
contributed a patch which adds a \texttt{__contains__()} magic method
for providing a custom implementation for \texttt{in}. Additionally, new built-in
objects written in C can define what \texttt{in} means for them via a new slot
in the sequence protocol.

Earlier versions of Python used a recursive algorithm for deleting
objects. Deeply nested data structures could cause the interpreter to
fill up the C stack and crash; Christian Tismer rewrote the deletion
logic to fix this problem. On a related note, comparing recursive
objects recursed infinitely and crashed; Jeremy Hylton rewrote the
code to no longer crash, producing a useful result instead. For
example, after this code:
\[
a = [] \\
b = []
\]
The comparison \texttt{a==b} returns true, because the two recursive data structures are isomorphic. See the thread “trashcan and PR#7” in the April 2000 archives of the python-dev mailing list for the discussion leading up to this implementation, and some useful relevant links. Note that comparisons can now also raise exceptions. In earlier versions of Python, a comparison operation such as \texttt{cmp(a,b)} would always produce an answer, even if a user-defined \texttt{__cmp__()} method encountered an error, since the resulting exception would simply be silently swallowed.

Work has been done on porting Python to 64-bit Windows on the Itanium processor, mostly by Trent Mick of ActiveState. (Confusingly, \texttt{sys.platform} is still \texttt{'win32'} on Win64 because it seems that for ease of porting, MS Visual C++ treats code as 32 bit on Itanium.) PythonWin also supports Windows CE; see the Python CE page at \url{http://pythonce.sourceforge.net/} for more information.

Another new platform is Darwin/MacOS X; initial support for it is in Python 2.0. Dynamic loading works, if you specify “configure –with-dyld –with-suffix=.x”. Consult the README in the Python source distribution for more instructions.

An attempt has been made to alleviate one of Python's warts, the often-confusing \texttt{NameError} exception when code refers to a local variable before the variable has been assigned a value. For example, the following code raises an exception on the \texttt{print} statement in both 1.5.2 and 2.0; in 1.5.2 a \texttt{NameError} exception is raised, while 2.0 raises a new \texttt{UnboundLocalError} exception. \texttt{UnboundLocalError} is a subclass of \texttt{NameError}, so any existing
code that expects NameError to be raised should still work.

```python
def f():
    print "i=", i
    i = i + 1
f()
```

Two new exceptions, TabError and IndentationError, have been introduced. They're both subclasses of SyntaxError, and are raised when Python code is found to be improperly indented.

**Changes to Built-in Functions**

A new built-in, `zip(seq1, seq2, ...)`, has been added. `zip()` returns a list of tuples where each tuple contains the i-th element from each of the argument sequences. The difference between `zip()` and `map(None, seq1, seq2)` is that `map()` pads the sequences with None if the sequences aren't all of the same length, while `zip()` truncates the returned list to the length of the shortest argument sequence.

The `int()` and `long()` functions now accept an optional “base” parameter when the first argument is a string. `int('123', 10)` returns 123, while `int('123', 16)` returns 291. `int(123, 16)` raises a TypeError exception with the message “can’t convert non-string with explicit base”.

A new variable holding more detailed version information has been added to the `sys` module. `sys.version_info` is a tuple `(major, minor, micro, level, serial)`. For example, in a hypothetical 2.0.1beta1, `sys.version_info` would be `(2, 0, 1, 'beta', 1)`. `level` is a string such as "alpha", "beta", or "final" for a
final release.

Dictionaries have an odd new method, `setdefault(key, default)`, which behaves similarly to the existing `get()` method. However, if the key is missing, `setdefault()` both returns the value of `default` as `get()` would do, and also inserts it into the dictionary as the value for `key`. Thus, the following lines of code:

```python
if dict.has_key(key): return dict[key]
else:
    dict[key] = []
return dict[key]
```

can be reduced to a single `return dict.setdefault(key, [])` statement.

The interpreter sets a maximum recursion depth in order to catch runaway recursion before filling the C stack and causing a core dump or GPF. Previously this limit was fixed when you compiled Python, but in 2.0 the maximum recursion depth can be read and modified using `sys.getrecursionlimit()` and `sys.setrecursionlimit()`. The default value is 1000, and a rough maximum value for a given platform can be found by running a new script, `Misc/find_recursionlimit.py`. 
Porting to 2.0

New Python releases try hard to be compatible with previous releases, and the record has been pretty good. However, some changes are considered useful enough, usually because they fix initial design decisions that turned out to be actively mistaken, that breaking backward compatibility can’t always be avoided. This section lists the changes in Python 2.0 that may cause old Python code to break.

The change which will probably break the most code is tightening up the arguments accepted by some methods. Some methods would take multiple arguments and treat them as a tuple, particularly various list methods such as append() and insert(). In earlier versions of Python, if L is a list, L.append(1,2) appends the tuple (1,2) to the list. In Python 2.0 this causes a TypeError exception to be raised, with the message: ‘append requires exactly 1 argument; 2 given’. The fix is to simply add an extra set of parentheses to pass both values as a tuple: L.append((1,2)).

The earlier versions of these methods were more forgiving because they used an old function in Python’s C interface to parse their arguments; 2.0 modernizes them to use PyArg_ParseTuple(), the current argument parsing function, which provides more helpful error messages and treats multi-argument calls as errors. If you absolutely must use 2.0 but can’t fix your code, you can edit Objects/listobject.c and define the preprocessor symbol NO_STRICT_LIST_APPEND to preserve the old behaviour; this isn’t recommended.

Some of the functions in the socket module are still forgiving in this way. For example, socket.connect(('hostname', 25)) is
the correct form, passing a tuple representing an IP address, but
socket.connect( 'hostname', 25 )() also works. socket.connect_ex() and socket.bind() are similarly easy-going. 2.0alpha1 tightened these functions up, but because the documentation actually used the erroneous multiple argument form, many people wrote code which would break with the stricter checking. GvR backed out the changes in the face of public reaction, so for the socket module, the documentation was fixed and the multiple argument form is simply marked as deprecated; it will be tightened up again in a future Python version.

The \x escape in string literals now takes exactly 2 hex digits. Previously it would consume all the hex digits following the ‘x’ and take the lowest 8 bits of the result, so \x123456 was equivalent to \x56.

The AttributeError and NameError exceptions have a more friendly error message, whose text will be something like Spam instance has no attribute 'eggs' or name 'eggs' is not defined. Previously the error message was just the missing attribute name eggs, and code written to take advantage of this fact will break in 2.0.

Some work has been done to make integers and long integers a bit more interchangeable. In 1.5.2, large-file support was added for Solaris, to allow reading files larger than 2 GiB; this made the tell() method of file objects return a long integer instead of a regular integer. Some code would subtract two file offsets and attempt to use the result to multiply a sequence or slice a string, but this raised a TypeError. In 2.0, long integers can be used to multiply or slice a sequence, and it'll behave as you'd intuitively expect it to; 3L * 'abc' produces 'abcabcabc', and (0,1,2,3)[2L:4L] produces (2,3). Long integers can also be used in various contexts where previously only integers
were accepted, such as in the `seek()` method of file objects, and in the formats supported by the `%` operator (%d, %i, %x, etc.). For example, "%d" % 2L**64 will produce the string 18446744073709551616.

The subtlest long integer change of all is that the `str()` of a long integer no longer has a trailing ‘L’ character, though `repr()` still includes it. The ‘L’ annoyed many people who wanted to print long integers that looked just like regular integers, since they had to go out of their way to chop off the character. This is no longer a problem in 2.0, but code which does `str(longval)[:-1]` and assumes the ‘L’ is there, will now lose the final digit.

Taking the `repr()` of a float now uses a different formatting precision than `str()`. `repr()` uses %.17g format string for C's `sprintf()`, while `str()` uses %.12g as before. The effect is that `repr()` may occasionally show more decimal places than `str()`, for certain numbers. For example, the number 8.1 can't be represented exactly in binary, so `repr(8.1)` is '8.0999999999999996', while `str(8.1)` is '8.1'.

The `-X` command-line option, which turned all standard exceptions into strings instead of classes, has been removed; the standard exceptions will now always be classes. The exceptions module containing the standard exceptions was translated from Python to a built-in C module, written by Barry Warsaw and Fredrik Lundh.
Extending/Embedding Changes

Some of the changes are under the covers, and will only be apparent to people writing C extension modules or embedding a Python interpreter in a larger application. If you aren’t dealing with Python’s C API, you can safely skip this section.

The version number of the Python C API was incremented, so C extensions compiled for 1.5.2 must be recompiled in order to work with 2.0. On Windows, it’s not possible for Python 2.0 to import a third party extension built for Python 1.5.x due to how Windows DLLs work, so Python will raise an exception and the import will fail.

Users of Jim Fulton’s ExtensionClass module will be pleased to find out that hooks have been added so that ExtensionClasses are now supported by `isinstance()` and `issubclass()`. This means you no longer have to remember to write code such as `if type(obj) == myExtensionClass`, but can use the more natural `if isinstance(obj, myExtensionClass)`.

The Python/importdl.c file, which was a mass of `#ifdefs` to support dynamic loading on many different platforms, was cleaned up and reorganised by Greg Stein. `importdl.c` is now quite small, and platform-specific code has been moved into a bunch of `Python/dynload_*.c` files. Another cleanup: there were also a number of `my*.h` files in the `Include/` directory that held various portability hacks; they’ve been merged into a single file, `Include/pyport.h`.

Vladimir Marangozov’s long-awaited malloc restructuring was completed, to make it easy to have the Python interpreter use a
custom allocator instead of C’s standard malloc(). For documentation, read the comments in Include/pymem.h and Include/objimpl.h. For the lengthy discussions during which the interface was hammered out, see the Web archives of the ‘patches’ and ‘python-dev’ lists at python.org.

Recent versions of the GUSI development environment for MacOS support POSIX threads. Therefore, Python’s POSIX threading support now works on the Macintosh. Threading support using the user-space GNU pth library was also contributed.

Threading support on Windows was enhanced, too. Windows supports thread locks that use kernel objects only in case of contention; in the common case when there’s no contention, they use simpler functions which are an order of magnitude faster. A threaded version of Python 1.5.2 on NT is twice as slow as an unthreaded version; with the 2.0 changes, the difference is only 10%. These improvements were contributed by Yakov Markovitch.

Python 2.0’s source now uses only ANSI C prototypes, so compiling Python now requires an ANSI C compiler, and can no longer be done using a compiler that only supports K&R C.

Previously the Python virtual machine used 16-bit numbers in its bytecode, limiting the size of source files. In particular, this affected the maximum size of literal lists and dictionaries in Python source; occasionally people who are generating Python code would run into this limit. A patch by Charles G. Waldman raises the limit from $2^{16}$ to $2^{32}$.

Three new convenience functions intended for adding constants to a module’s dictionary at module initialization time were added: PyModule_AddObject(), PyModule_AddIntConstant(), and
PyModule_AddStringConstant(). Each of these functions takes a module object, a null-terminated C string containing the name to be added, and a third argument for the value to be assigned to the name. This third argument is, respectively, a Python object, a C long, or a C string.

A wrapper API was added for Unix-style signal handlers. PyOS_getsig() gets a signal handler and PyOS_setsig() will set a new handler.
Distutils: Making Modules Easy to Install

Before Python 2.0, installing modules was a tedious affair – there was no way to figure out automatically where Python is installed, or what compiler options to use for extension modules. Software authors had to go through an arduous ritual of editing Makefiles and configuration files, which only really work on Unix and leave Windows and MacOS unsupported. Python users faced wildly differing installation instructions which varied between different extension packages, which made administering a Python installation something of a chore.

The SIG for distribution utilities, shepherded by Greg Ward, has created the Distutils, a system to make package installation much easier. They form the distutils package, a new part of Python's standard library. In the best case, installing a Python module from source will require the same steps: first you simply mean unpack the tarball or zip archive, and the run “python setup.py install”. The platform will be automatically detected, the compiler will be recognized, C extension modules will be compiled, and the distribution installed into the proper directory. Optional command-line arguments provide more control over the installation process, the distutils package offers many places to override defaults – separating the build from the install, building or installing in non-default directories, and more.

In order to use the Distutils, you need to write a setup.py script. For the simple case, when the software contains only .py files, a minimal setup.py can be just a few lines long:

```python
from distutils.core import setup
setup (name = "foo", version = "1.0",
        py_modules = ["module1", "module2"])
```
The `setup.py` file isn't much more complicated if the software consists of a few packages:

```python
from distutils.core import setup
setup (name = "foo", version = "1.0",
       packages = ["package", "package.subpackage"])
```

A C extension can be the most complicated case; here's an example taken from the PyXML package:

```python
from distutils.core import setup, Extension

expat_extension = Extension('xml.parsers.pyexpat',
                            define_macros = [(['XML_NS', None]),
                                            include_dirs = ['extensions/expat/xmltok',
                                                            'extensions/expat/xmlparse'],
                                            sources = ['extensions/pyexpat.c',
                                                      'extensions/expat/xmltok/xmltok.c',
                                                      'extensions/expat/xmltok/xmlrole.c'],
                            )
setup (name = "PyXML", version = "0.5.4",
       ext_modules = [ expat_extension ] )
```

The Distutils can also take care of creating source and binary distributions. The “sdist” command, run by “python setup.py sdist”, builds a source distribution such as `foo-1.0.tar.gz`. Adding new commands isn’t difficult, “bdist_rpm” and “bdist_wininst” commands have already been contributed to create an RPM distribution and a Windows installer for the software, respectively. Commands to create other distribution formats such as Debian packages and Solaris `.pkg` files are in various stages of development.
All this is documented in a new manual, *Distributing Python Modules*, that joins the basic set of Python documentation.
XML Modules

Python 1.5.2 included a simple XML parser in the form of the `xml` module, contributed by Sjoerd Mullender. Since 1.5.2's release, two different interfaces for processing XML have become common: SAX2 (version 2 of the Simple API for XML) provides an event-driven interface with some similarities to `xml`, and the DOM (Document Object Model) provides a tree-based interface, transforming an XML document into a tree of nodes that can be traversed and modified. Python 2.0 includes a SAX2 interface and a stripped-down DOM interface as part of the `xml` package. Here we will give a brief overview of these new interfaces; consult the Python documentation or the source code for complete details. The Python XML SIG is also working on improved documentation.

SAX2 Support

SAX defines an event-driven interface for parsing XML. To use SAX, you must write a SAX handler class. Handler classes inherit from various classes provided by SAX, and override various methods that will then be called by the XML parser. For example, the `startElement()` and `endElement()` methods are called for every starting and end tag encountered by the parser, the `characters()` method is called for every chunk of character data, and so forth.

The advantage of the event-driven approach is that the whole document doesn't have to be resident in memory at any one time, which matters if you are processing really huge documents. However, writing the SAX handler class can get very complicated if you're trying to modify the document structure in some elaborate way.
For example, this little example program defines a handler that prints a message for every starting and ending tag, and then parses the file hamlet.xml using it:

```python
from xml import sax

class SimpleHandler(sax.ContentHandler):
    def startElement(self, name, attrs):
        print 'Start of element:', name, attrs.keys()

    def endElement(self, name):
        print 'End of element:', name

# Create a parser object
parser = sax.make_parser()

# Tell it what handler to use
handler = SimpleHandler()
parsed.setContentHandler( handler )

# Parse a file!
parsed.parse( 'hamlet.xml' )
```

For more information, consult the Python documentation, or the XML HOWTO at http://pyxml.sourceforge.net/topics/howto/xml-howto.html.

**DOM Support**

The Document Object Model is a tree-based representation for an XML document. A top-level Document instance is the root of the tree, and has a single child which is the top-level Element instance. This Element has children nodes representing character data and any sub-elements, which may have further children of their own, and so
forth. Using the DOM you can traverse the resulting tree any way you like, access element and attribute values, insert and delete nodes, and convert the tree back into XML.

The DOM is useful for modifying XML documents, because you can create a DOM tree, modify it by adding new nodes or rearranging subtrees, and then produce a new XML document as output. You can also construct a DOM tree manually and convert it to XML, which can be a more flexible way of producing XML output than simply writing `<tag1>...<tag1>` to a file.

The DOM implementation included with Python lives in the `xml.dom.minidom` module. It's a lightweight implementation of the Level 1 DOM with support for XML namespaces. The `parse()` and `parseString()` convenience functions are provided for generating a DOM tree:

```python
from xml.dom import minidom
doc = minidom.parse('hamlet.xml')
```

doc is a Document instance. Document, like all the other DOM classes such as Element and Text, is a subclass of the Node base class. All the nodes in a DOM tree therefore support certain common methods, such as `toxml()` which returns a string containing the XML representation of the node and its children. Each class also has special methods of its own; for example, Element and Document instances have a method to find all child elements with a given tag name. Continuing from the previous 2-line example:

```python
perslist = doc.getElementsByTagName('PERSONA')
print perslist[0].toxml()
print perslist[1].toxml()
```
For the *Hamlet* XML file, the above few lines output:

```xml
<PERSONA>CLAUDIUS, king of Denmark. </PERSONA>
<PERSONA>HAMLET, son to the late, and nephew to the
```

The root element of the document is available as `doc.documentElement`, and its children can be easily modified by deleting, adding, or removing nodes:

```python
root = doc.documentElement

# Remove the first child
root.removeChild( root.childNodes[0] )

# Move the new first child to the end
root.appendChild( root.childNodes[0] )

# Insert the new first child (originally, the third child) before the 20th child.
root.insertBefore( root.childNodes[0], root.childNodes[19] )
```

Again, I will refer you to the Python documentation for a complete listing of the different Node classes and their various methods.

**Relationship to PyXML**

The XML Special Interest Group has been working on XML-related Python code for a while. Its code distribution, called PyXML, is available from the SIG's Web pages at [https://www.python.org/community/sigs/current/xml-sig](https://www.python.org/community/sigs/current/xml-sig). The PyXML distribution also used the package name *xml*. If you’ve written programs that used PyXML, you’re probably wondering about its
compatibility with the 2.0 xml package.

The answer is that Python 2.0’s xml package isn’t compatible with PyXML, but can be made compatible by installing a recent version PyXML. Many applications can get by with the XML support that is included with Python 2.0, but more complicated applications will require that the full PyXML package will be installed. When installed, PyXML versions 0.6.0 or greater will replace the xml package shipped with Python, and will be a strict superset of the standard package, adding a bunch of additional features. Some of the additional features in PyXML include:

- 4DOM, a full DOM implementation from FourThought, Inc.
- The xmlproc validating parser, written by Lars Marius Garshol.
- The sgmlop parser accelerator module, written by Fredrik Lundh.
Module changes

Lots of improvements and bugfixes were made to Python’s extensive standard library; some of the affected modules include readline, ConfigParser, cgi, calendar, posix, readline, xmllib, aifc, chunk, wave, random, shelve, and nntplib. Consult the CVS logs for the exact patch-by-patch details.

Brian Gallew contributed OpenSSL support for the socket module. OpenSSL is an implementation of the Secure Socket Layer, which encrypts the data being sent over a socket. When compiling Python, you can edit Modules/Setup to include SSL support, which adds an additional function to the socket module: socket.ssl(socket, keyfile, certfile), which takes a socket object and returns an SSL socket. The httplib and urllib modules were also changed to support https:// URLs, though no one has implemented FTP or SMTP over SSL.

The httplib module has been rewritten by Greg Stein to support HTTP/1.1. Backward compatibility with the 1.5 version of httplib is provided, though using HTTP/1.1 features such as pipelining will require rewriting code to use a different set of interfaces.

The Tkinter module now supports Tcl/Tk version 8.1, 8.2, or 8.3, and support for the older 7.x versions has been dropped. The Tkinter module now supports displaying Unicode strings in Tk widgets. Also, Fredrik Lundh contributed an optimization which makes operations like create_line and create_polygon much faster, especially when using lots of coordinates.

The curses module has been greatly extended, starting from Oliver
Andrich's enhanced version, to provide many additional functions from ncurses and SYSV curses, such as colour, alternative character set support, pads, and mouse support. This means the module is no longer compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

As mentioned in the earlier discussion of 2.0's Unicode support, the underlying implementation of the regular expressions provided by the re module has been changed. SRE, a new regular expression engine written by Fredrik Lundh and partially funded by Hewlett Packard, supports matching against both 8-bit strings and Unicode strings.
New modules

A number of new modules were added. We’ll simply list them with brief descriptions; consult the 2.0 documentation for the details of a particular module.

- **atexit**: For registering functions to be called before the Python interpreter exits. Code that currently sets `sys.exitfunc` directly should be changed to use the `atexit` module instead, importing `atexit` and calling `atexit.register()` with the function to be called on exit. (Contributed by Skip Montanaro.)
- **codecs**, **encodings**, **unicodedata**: Added as part of the new Unicode support.
- **filecmp**: Supersedes the old `cmp`, `cmpcache` and `dircmp` modules, which have now become deprecated. (Contributed by Gordon MacMillan and Moshe Zadka.)
- **gettext**: This module provides internationalization (I18N) and localization (L10N) support for Python programs by providing an interface to the GNU gettext message catalog library. (Integrated by Barry Warsaw, from separate contributions by Martin von Löwis, Peter Funk, and James Henstridge.)
- **linuxaudiodev**: Support for the `/dev/audio` device on Linux, a twin to the existing `sunaudiodev` module. (Contributed by Peter Bosch, with fixes by Jeremy Hylton.)
- **mmap**: An interface to memory-mapped files on both Windows and Unix. A file’s contents can be mapped directly into memory, at which point it behaves like a mutable string, so its contents can be read and modified. They can even be passed to functions that expect ordinary strings, such as the `re` module. (Contributed by Sam Rushing, with some extensions by A.M. Kuchling.)
- **pyexpat**: An interface to the Expat XML parser. (Contributed by Paul Prescod.)
robotparser: Parse a robots.txt file, which is used for writing Web spiders that politely avoid certain areas of a Web site. The parser accepts the contents of a robots.txt file, builds a set of rules from it, and can then answer questions about the fetchability of a given URL. (Contributed by Skip Montanaro.)

tabnanny: A module/script to check Python source code for ambiguous indentation. (Contributed by Tim Peters.)

UserString: A base class useful for deriving objects that behave like strings.

webbrowser: A module that provides a platform independent way to launch a web browser on a specific URL. For each platform, various browsers are tried in a specific order. The user can alter which browser is launched by setting the BROWSER environment variable. (Originally inspired by Eric S. Raymond's patch to urllib which added similar functionality, but the final module comes from code originally implemented by Fred Drake as Tools/idle/BrowserControl.py, and adapted for the standard library by Fred.)

_winreg: An interface to the Windows registry. _winreg is an adaptation of functions that have been part of PythonWin since 1995, but has now been added to the core distribution, and enhanced to support Unicode. _winreg was written by Bill Tutt and Mark Hammond.

zipfile: A module for reading and writing ZIP-format archives. These are archives produced by PKZIP on DOS/Windows or zip on Unix, not to be confused with gzip-format files (which are supported by the gzip module) (Contributed by James C. Ahlstrom.)
imputil: A module that provides a simpler way for writing customized import hooks, in comparison to the existing ihooks module. (Implemented by Greg Stein, with much discussion on python-dev along the way.)
IDLE Improvements

IDLE is the official Python cross-platform IDE, written using Tkinter. Python 2.0 includes IDLE 0.6, which adds a number of new features and improvements. A partial list:

- UI improvements and optimizations, especially in the area of syntax highlighting and auto-indentation.
- The class browser now shows more information, such as the top level functions in a module.
- Tab width is now a user settable option. When opening an existing Python file, IDLE automatically detects the indentation conventions, and adapts.
- There is now support for calling browsers on various platforms, used to open the Python documentation in a browser.
- IDLE now has a command line, which is largely similar to the vanilla Python interpreter.
- Call tips were added in many places.
- IDLE can now be installed as a package.
- In the editor window, there is now a line/column bar at the bottom.
- Three new keystroke commands: Check module (Alt-F5), Import module (F5) and Run script (Ctrl-F5).
Deleted and Deprecated Modules

A few modules have been dropped because they’re obsolete, or because there are now better ways to do the same thing. The stdwin module is gone; it was for a platform-independent windowing toolkit that’s no longer developed.

A number of modules have been moved to the lib-old subdirectory: cmp, cmpcache, dircmp, dump, find, grep, packmail, poly, util, whatsound, zmod. If you have code which relies on a module that’s been moved to lib-old, you can simply add that directory to sys.path to get them back, but you’re encouraged to update any code that uses these modules.
Acknowledgements

The authors would like to thank the following people for offering suggestions on various drafts of this article: David Bolen, Mark Hammond, Gregg Hauser, Jeremy Hylton, Fredrik Lundh, Detlef Lannert, Aahz Maruch, Skip Montanaro, Vladimir Marangozov, Tobias Polzin, Guido van Rossum, Neil Schemenauer, and Russ Schmidt.
Changelog

The NEWS file is not available.
1. Whetting Your Appetite

If you do much work on computers, eventually you find that there’s some task you’d like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you’d like to write a small custom database, or a specialized GUI application, or a simple game.

If you’re a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you’re writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you’ve written a program that could use an extension language, and you don’t want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, Mac OS X, and Unix operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and
dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the
Python interpreter into an application written in C and use it as an extension or command language for that application.

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you’ll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.
2. Using the Python Interpreter
2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.7` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```bash
python3.7
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python37`, though you can change this when you’re running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python37
```

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support readline. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input.
**Editing and History Substitution** for an introduction to the keys. If nothing appears to happen, or if \^P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety with single quotes.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command line options are described in Command line and environment.

### 2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to
the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-` (meaning standard input), `sys.argv[0]` is set to `'-'.` When `-c command` is used, `sys.argv[0]` is set to `'-c'.` When `-m module` is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c command` or `-m module` are not consumed by the Python interpreter’s option processing but left in `sys.argv` for the command or module to handle.

**2.1.2. Interactive Mode**

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (`>>>`); for continuation lines it prompts with the secondary prompt, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.7
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> the_world_is_flat = True
>>> if the_world_is_flat:
```
... print("Be careful not to fall off!")
...
Be careful not to fall off!

For more on interactive mode, see Interactive Mode.
2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the first line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where `encoding` is one of the valid `codecs` supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the first line rule is when the source code starts with a UNIX “shebang” line. In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```
Footnotes

On Unix, the Python 3.x interpreter is by default not installed with the executable named python, so that it does not conflict with a simultaneously installed Python 2.x executable.

[1]
3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1  # and this is the second comment
    # ... and now a third!
text = "# This is not a comment because it's inside
```
3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point
1.6
```

The integer numbers (e.g. 2, 4, 20) have type `int`, the ones with a fractional part (e.g. 5.0, 1.6) have type `float`. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%:

```
>>> 17 / 3  # classic division returns a float
```
With Python, it is possible to use the ** operator to calculate powers [1]:

```python
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```python
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not "defined" (assigned a value), trying to use it will give you an error:

```python
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type
operands convert the integer operand to floating point:

```python
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```python
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for `complex numbers`, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

### 3.1.2. Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2]. `\` can be
used to escape quotes:

```python
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote.
'doesn't'
>>> "doesn't"  # ...or use double quotes instead
'doesn't'
>>> '''Yes," they said.'
'"Yes," they said.'
>>> ""Yes," they said."
'Yes," they said.'
>>> ""Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```python
>>> ""Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print(""Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```
If you don’t want characters preaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

```python
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

```python
print(""
Usage: thingy [OPTIONS]
   -h                                Display this usage message
   -H hostname                        Hostname to connect to
"")
```

produces the following output (note that the initial newline is not included):

```python
Usage: thingy [OPTIONS]
   -h                                Display this usage message
   -H hostname                        Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```python
>>> # 3 times 'un', followed by 'ium'
```
Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```python
>>> 3 * 'un' + 'ium'
'unununium'
```

This feature is particularly useful when you want to break long strings:

```python
>>> text = ('Put several strings within parentheses ...
            to have them joined together.'
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```python
>>> prefix = 'Py'
>>> prefix + 'thon'
```

```python
SyntaxError: invalid syntax
```

```python
>>> (un' * 3) 'ium'
File "<stdin>", line 1
   (un' * 3) 'ium'
```

```python
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use +:

```python
>>> prefix + 'thon'
```
Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```python
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```python
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]  #
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```python
>>> word[0:2]  # characters from position 0 (include)
'Py'
>>> word[2:5]  # characters from position 2 (include)
'tho'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`: 
Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```python
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0 1 2 3 4 5 6
-6 -5 -4 -3 -2 -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of
word[1:3] is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'

>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'

>>> word[:2] + 'py'
'Pypy'
```
The built-in function `len()` returns the length of a string:

```python
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also:

**Text Sequence Type — str**
Strings are examples of sequence types, and support the common operations supported by such types.

**String Methods**
Strings support a large number of methods for basic transformations and searching.

**Formatted string literals**
String literals that have embedded expressions.

**Format String Syntax**
Information about string formatting with `str.format()`.

**printf-style String Formatting**
The old formatting operations invoked when strings are the left operand of the `%` operator are described in more detail here.

### 3.1.3. Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the `list`, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.
Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```python
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```python
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```python
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```python
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```
You can also add new items at the end of the list, by using the `append()` method (we will see more about methods later):

```python
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```python
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list:
>>> letters[:]=[]
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```python
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```
It is possible to nest lists (create lists containing other lists), for example:

```python
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```
3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```python
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
... 0
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a multiple assignment: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The while loop executes as long as the condition (here: `a < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list
value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

- The body of the loop is *indented*: indentation is Python’s way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```python
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

The keyword argument `end` can be used to avoid the newline after the output, or end the output with a different string:

```python
>>> a, b = 0, 1
>>> while a < 1000:
```
```python
... print(a, end=',',)
...
... a, b = b, a+b
...
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

### Footnotes

Since `**` has higher precedence than `-`, `-3**2` will be interpreted as `- (3**2)` and thus result in `-9`. To avoid this and get `9`, you can use `(-3)**2`.

Unlike other languages, special characters such as `\n` have the same meaning with both single (`'...'`) and double (`"..."`) quotes. The only difference between the two is that within single quotes you don't need to escape `"` (but you have to escape `\`) and vice versa.
4. More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.
4.1. if Statements

Perhaps the most well-known statement type is the if statement. For example:

```python
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...    x = 0
...    print('Negative changed to zero')
...    elif x == 0:
...        print('Zero')
...        elif x == 1:
...            print('Single')
...            else:
...                print('More')
                    More
```

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.
4.2. for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```python
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...  cat    3
... window  6
... defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```python
>>> for w in words[:]: # Loop over a slice copy of
...     if len(w) > 6:
...         words.insert(0, w)
... >>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

With for w in words:, the example would attempt to create an
infinite list, inserting defenestrate over and over again.
4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```python
>>> for i in range(5):
    ...     print(i)
... 0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```python
range(5, 10)  
    5, 6, 7, 8, 9  

range(0, 10, 3)  
    0, 3, 6, 9  

range(-10, -100, -30)  
    -10, -40, -70  
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:
```python
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
... 0 Mary 1 had 2 a 3 little 4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see Looping Techniques.

A strange thing happens if you just print a range:

```python
>>> print(range(10))
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such an iterator. The function `list()` is another; it creates lists from iterables:

```python
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Later we will see more functions that return iterables and take iterables as argument.
4.4. break and continue Statements, and else Clauses on Loops

The `break` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```python
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, not the `if` statement.)
When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a try statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see Handling Exceptions.

The `continue` statement, also borrowed from C, continues with the next iteration of the loop:

```python
>>> for num in range(2, 10):
    ...    if num % 2 == 0:
    ...        print("Found an even number", num)
    ...    continue
    ...    print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```
4.5. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```python
>>> while True:
    ...    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
    ...
```

This is commonly used for creating minimal classes:

```python
>>> class MyEmptyClass:
    ...    pass
    ...
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```python
>>> def initlog(*args):
    ...    pass  # Remember to implement this!
    ...
```
4.6. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```python
>>> def fib(n):  # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
...     fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function `definition`. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or `docstring`. (More about docstrings can be found in the section `Documentation Strings`.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The `execution` of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments
in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn’t return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it’s a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:
It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```python
>>> def fib2(n):
...     # return Fibonacci series up to n
...     '''Return a list containing the Fibonacci series up to n.'''
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)  # see below
...         a, b = b, a+b
...     return result
... >>> f100 = fib2(100)  # call it
... >>> f100
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a `method` of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using `classes`, see Classes) The method `append()` shown in the example is defined for list.
objects; it adds a new element at the end of the list. In this example it is equivalent to \texttt{result = result + [a]}, but more efficient.
4.7. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```python
def ask_ok(prompt, retries=4, reminder='Please try again! while True:
    ok = input(prompt)
    if ok in ('y', 'ye', 'yes'):
        return True
    if ok in ('n', 'no', 'nop', 'nope'):
        return False
    retries = retries - 1
    if retries < 0:
        raise ValueError('invalid user response')
    print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`
This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```python
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

will print 5.

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```python
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```
If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

```python
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2. Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```python
def parrot(voltage, state='a stiff', action='voom',
    print("-- This parrot wouldn't", action, end=''
    print("if you put", voltage, "volts through it.
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```python
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='VOOOOOM')
parrot(action='VOOOOOM', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

but all the following calls would be invalid:
In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

```python
>>> def function(a):
    ...    pass
    ...

>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see Mapping Types — `dict`) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```python
def cheeseshop(kind, *arguments, **keywords):
```
print("-- Do you have any", kind, "?"
print("-- I'm sorry, we're all out of", kind)
for arg in arguments:
    print(arg)
print("-" * 40)
for kw in keywords:
    print(kw, ":", keywords[kw])

It could be called like this:

cheeseshop("Limburger", "It's very runny, sir.",
    "It's really very, VERY runny, sir.",
    shopkeeper="Michael Palin",
    client="John Cleese",
    sketch="Cheese Shop Sketch")

and of course it would print:

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.7.3. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will
be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

```python
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the *args parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

```python
>>> def concat(*args, sep=":"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

### 4.7.4. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in range() function expects separate start and stop arguments. If they are not available separately, write the function call with the *-operator to unpack the arguments out of a list or tuple:

```python
>>> list(range(3, 6))
# normal call with
[3, 4, 5]
>>> args = [3, 6]
```
In the same fashion, dictionaries can deliver keyword arguments with the **-operator:

```python
>>> def parrot(voltage, state='a stiff', action='voom', **d):
...    print("-- This parrot wouldn't", action, end"
...    print("if you put", voltage, "volts through it.
...    print("E's", state, "!")

>>> d = {'voltage': "four million", 'state': "bleedin"
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million

4.7.5. Lambda Expressions

Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```python
>>> def make_incrementor(n):
...    return lambda x: x + n

>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```
The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```python
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'),
            (4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.)
Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```python
>>> def my_function():
...     """Do nothing, but document it.
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.
    No, really, it doesn't do anything.
```

4.7.7. Function Annotations

Function annotations are completely optional metadata information about the types used by user-defined functions (see PEP 3107 and PEP 484 for more information).

Annotations are stored in the __annotations__ attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal ->, followed by an expression, between the parameter list and the colon denoting the end of the def statement. The following example has a positional
argument, a keyword argument, and the return value annotated:

```python
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...    print("Annotations:", f.__annotations__)
...    print("Arguments:", ham, eggs)
...    return ham + ' and ' + eggs

>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```
4.8. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about coding style. Most languages can be written (or more concise, formatted) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, PEP 8 has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
  
  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don’t exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly
inside bracketing constructs: \( a = f(1, 2) + g(3, 4) \).

- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see *A First Look at Classes* for more on classes and methods).

- Don’t use fancy encodings if your code is meant to be used in international environments. Python’s default, UTF-8, or even plain ASCII work best in any case.

- Likewise, don’t use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

**Footnotes**

Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).
5. Data Structures

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.
5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.append(x)`
Add an item to the end of the list. Equivalent to `a[len(a):] = [x].`

`list.extend(iterable)`
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`. 

`list.insert(i, x)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x) is equivalent to a.append(x).`

`list.remove(x)`
Remove the first item from the list whose value is equal to x. It raises a `ValueError` if there is no such item.

`list.pop([i])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
list.**clear**()  
Remove all items from the list. Equivalent to del a[:].

list.**index**(x[, start[, end]])  
Return zero-based index in the list of the first item whose value is equal to x. Raises a **ValueError** if there is no such item.

   The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

list.**count**(x)  
Return the number of times x appears in the list.

list.**sort**(key=None, reverse=False)  
Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.**reverse**()  
Reverse the elements of the list in place.

list.**copy**()  
Return a shallow copy of the list. Equivalent to a[:].

An example that uses most of the list methods:

```python
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
```
You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()` method. To retrieve an item from the top of the stack, use `pop()` method without an explicit index. For example:

```python
>>> stack = [3, 4, 5]  
>>> stack.append(6)  
>>> stack.append(7)  
>>> stack            
[3, 4, 5, 6, 7]  
>>> stack.pop()    
7
```
5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```python
>>> stack = [3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

```python
>>> from collections import deque
>>> queue = deque(['Eric', 'John', 'Michael'])
>>> queue.append('Terry')        # Terry arrives
>>> queue.append('Graham')       # Graham arrives
>>> queue.popleft()              # The first to arrive now leaves
'Eric'
>>> queue.popleft()              # The second to arrive now leaves
'John'
>>> queue                     # Remaining queue
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. List Comprehensions
List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```python
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```python
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```python
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```python
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```
and it's equivalent to:

```python
>>> combs = []
>>> for x in [1, 2, 3):
...    for y in [3, 1, 4]:
...        if x != y:
...            combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the `for` and `if` statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```python
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['banana', 'loganberry', 'passion fruit']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```
```python
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
    ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```python
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

### 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```python
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

The following list comprehension will transpose rows and columns:
As we saw in the previous section, the nested listcomp is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```python
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```python
>>> transposed = []
>>> for i in range(4):
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case:

```python
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See [Unpacking Argument Lists](#) for details on the asterisk in this line.
5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```python
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```python
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.
5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data types (see Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the tuple.

A tuple consists of a number of values separated by commas, for instance:

```python
>>> t = 12345, 54321, 'hello!
```nn
As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are
necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of namedtuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```python
>>> empty = ()
>>> singleton = 'hello',  # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```python
>>> x, y, z = t
```
This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.
5.4. Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```python
>>> basket = {'apple', 'orange', 'apple', 'pear', 'cranberry', 'orange'}
>>> print(basket)  # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket  # fast membership testing
True
>>> 'crabgrass' in basket
False
>>> # Demonstrate set operations on unique letters from two words
... >>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b  # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b  # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b  # letters in both a and b
```

Similarly to list comprehensions, set comprehensions are also supported:

```python
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
dict
{'r', 'd'}
```
5.5. Dictionaries

Another useful data type built into Python is the dictionary (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the in keyword.
Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([(sape, 4139), ('guido', 4127), ('jack',
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
5.6. Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```python
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...     gallahad the pure
    robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```python
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...     0 tic
    1 tac
    2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```python
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...     What is your name? It is lancelot.
    What is your quest? It is the holy grail.
```
What is your favorite color? It is blue.

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```python
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```python
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```python
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
```
... filtered_data.append(value)
...

>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
5.7. More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operators is and is not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, \( a < b == c \) tests whether \( a \) is less than \( b \) and moreover \( b \) equals \( c \).

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that \( A \) and not \( B \) or \( C \) is equivalent to \( (A \) and (not \( B \))) \) or \( C \). As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if \( A \) and \( C \) are true but \( B \) is false, \( A \) and \( B \) and \( C \) does not evaluate the expression \( C \). When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean
expression to a variable. For example,

```python
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.
5.8. Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical ordering*: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

\[
\begin{align*}
(1, 2, 3) & < (1, 2, 4) \\
[1, 2, 3] & < [1, 2, 4] \\
'ABC' & < 'C' < 'Pascal' < 'Python' \\
(1, 2, 3, 4) & < (1, 2, 4) \\
(1, 2) & < (1, 2, -1) \\
(1, 2, 3) & \equiv (1.0, 2.0, 3.0) \\
(1, 2, ('aa', 'ab')) & < (1, 2, ('abc', 'a'), 4)
\end{align*}
\]

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.
Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.
6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

```python
# Fibonacci numbers module
def fib(n):  # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
```
a, b = b, a+b
print()

def fib2(n):
    # return Fibonacci series up to n
    result = []
a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

Now enter the Python interpreter and import this module with the following command:

```python
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```python
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```python
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user’s global variables. On the other hand, if you know what you are doing you can touch a module’s global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module’s global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module’s symbol table. For example:

```python
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).
There is even a variant to import all names that a module defines:

```python
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by as, then the name following as is bound directly to the imported module.

```python
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that import fibo will do, with the only difference of it being available as fib.

It can also be used when utilising from with similar effects:

```python
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**Note:** For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to
6.1.1. Executing modules as scripts

When you run a Python module with

```python
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to "__main__". That means that by adding this code at the end of your module:

```python
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo

>>> 
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).
6.1.2. The Module Search Path

When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable sys.path. sys.path is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- The installation-dependent default.

**Note:** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is not added to the module search path.

After initialization, Python programs can modify sys.path. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section Standard Modules for more information.

6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the __pycache__ directory under the name module.version.pyc, where the version encodes the format of the compiled file; it generally contains the Python version number. For
example, in CPython release 3.3 the compiled version of spam.py would be cached as __pycache__/spam.cpython-33.pyc. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the -O or -OO switches on the Python command to reduce the size of a compiled module. The -O switch removes assert statements, the -OO switch removes both assert statements and __doc__ strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. “Optimized” modules have an opt- tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn’t run any faster when it is read from a .pyc file than when it is read from a .py file; the only thing that's faster about .pyc files is the speed with which they are loaded.
- The module compileall can create .pyc files for all modules in a directory.
• There is more detail on this process, including a flow chart of the decisions, in PEP 3147.
6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```python
>>> import sys
>>> sys.ps1
'>>>
>>> sys.ps2
'...
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list
operations:

```python
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```
6.3. The \texttt{dir()} Function

The built-in function \texttt{dir()} is used to find out which names a module defines. It returns a sorted list of strings:

\begin{verbatim}
>>> import fibo, sys
>>> dir(fibo)  
['__name__', 'fib', 'fib2']
>>> dir(sys)
\end{verbatim}

Without arguments, \texttt{dir()} lists the names you have defined currently:

\begin{verbatim}
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
\end{verbatim}
Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```python
>>> import builtins
>>> dir(builtins)
```
['sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
6.4. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `wav`, `aiff`, `au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/
    __init__.py
    formats/
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
```

Top-level package
Initialize the sound package
Subpackage for file formats
When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```python
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.
An alternative way of importing the submodule is:

```python
from sound.effects import echo
```

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

```python
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```python
from sound.effects.echo import echofilter
```

Again, this loads the submodule echo, but this makes its function `echofilter()` directly available:

```python
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can’t be a class or function or variable defined in the previous item.
6.4.1. Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing submodules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ['echo', 'surround', 'reverse']
```

This would mean that `from sound.effects import *` would import the three named submodules of the sound package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and
submodules explicitly loaded) by __init__.py. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

```python
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the echo and surround modules are imported in the current namespace because they are defined in the sound.effects package when the from...import statement is executed. (This also works when __all__ is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practice in production code.

Remember, there is nothing wrong with using from Package import specific_submodule! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

### 6.4.2. Intra-package References

When packages are structured into subpackages (as with the sound package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module sound.filters.vocoder needs to use the echo module in the sound.effects package, it can use from sound.effects import echo.

You can also write relative imports, with the from module import
name form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the surround module for example, you might use:

```python
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application must always use absolute imports.

### 6.4.3. Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

#### Footnotes

[1] In fact function definitions are also ‘statements’ that are ‘executed’; the execution of a module-level function definition enters the function name in the module’s global symbol table.
7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.
7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: expression statements and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use formatted string literals, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

  ```python
  >>> year = 2016
  >>> event = 'Referendum'
  >>> f'Results of the {year} {event}'
  'Results of the 2016 Referendum'
  ```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.

  ```python
  >>> yes_votes = 42_572_654
  >>> no_votes = 43_132_495
  >>> percentage = yes_votes / (yes_votes + no_votes)
  >>> '{:-9} YES votes {:2.2%}'.format(yes_votes,
  ...   '42572654 YES votes 49.67%'
  ```
• Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don’t need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don’t have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```python
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y)
>>> print(s)
The value of x is 32.5, and y is 400000...
>>> # The repr() of a string adds string quotes and
```
The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```python
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}
The value of pi is approximately 3.142.
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
```
Other modifiers can be used to convert the value before it is formatted. '!'a' applies ascii(), '!'s' applies str(), and '!'r' applies repr():

```python
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

For a reference on these format specifications, see the reference guide for the Format Specification Mini-Language.

### 7.1.2. The String format() Method

Basic usage of the `str.format()` method looks like this:

```python
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni!'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```python
>>> print('{}0 and {}1'.format('spam', 'eggs'))
```
spam and eggs

```python
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```python
>>> print('This {food} is {adjective}.'.format...
...     food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```python
>>> print('The story of {0}, {1}, and {other}.'.format...
...     other='Georg'))
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don’t want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; ... Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the `**` notation.

```python
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: ...
...     Jack: 4098; Sjoerd: 4127; Dcab: 8637678
This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes:

```python
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x),
...     end=''
...     )
... print(repr(x*x*x).rjust(4))
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see `Format String Syntax`.

### 7.1.3. Manual String Formatting

Here’s the same table of squares and cubes, formatted manually:

```python
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3),
...    end='
...     # Note use of 'end' on previous line
...     ')
... print(repr(x*x*x).rjust(4))
1 1 1
<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don’t truncate it, but return it unchanged; this will mess up your column lay-out but that’s usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[::n].`)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```python
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. Old string formatting
The `%` operator can also be used for string formatting. It interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

```python
>>> import math
>>> print('The value of pi is approximately %5.3f.'
The value of pi is approximately 3.142.
```

More information can be found in the `printf-style String Formatting` section.
7.2. Reading and Writing Files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```python
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The `mode` argument is optional; `'r'` will be assumed if it’s omitted.

Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see `open()`). `'b'` appended to the mode opens the file in binary mode: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.
It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```python
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

If you’re not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it. If you don’t explicitly close a file, Python’s garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```python
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

### 7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some
quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most size bytes are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

```python
>>> f.read()
'This is the entire file.\n'
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```python
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```python
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
```
If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

f.write(string) writes the contents of string to the file, returning the number of characters written.

```python
>>> f.write('This is a test
')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```python
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

f.tell() returns an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object’s position, use f.seek(offset, from_what). The position is computed from adding offset to a reference point; the reference point is selected by the from_what argument. A from_what value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. from_what can be omitted and defaults to 0, using the beginning of the file as the reference point.

```python
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
```
In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)` and the only valid offset values are those returned from the `f.tell()`, or zero. Any other offset value produces undefined behaviour.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

### 7.2.2. Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**. The standard module called `json` can take Python data
Hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

**Note:** The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object \( x \), you can view its JSON string representation with a simple line of code:

```python
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a text file. So if \( f \) is a text file object opened for writing, we can do this:

```python
json.dump(x, f)
```

To decode the object again, if \( f \) is a text file object which has been opened for reading:

```python
x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of
Contrary to JSON, pickle is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

See also: pickle - the pickle module
8. Errors and Exceptions

Until now error messages haven’t been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.
8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```python
>>> while True print('Hello world')
File "<stdin>", line 1
    while True print('Hello world')
      ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function `print()`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.
8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and
what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

**Built-in Exceptions** lists the built-in exceptions and their meanings.
8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the **KeyboardInterrupt** exception.

```python
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number.")
...```

The **try** statement works as follows.

- First, the **try clause** (the statement(s) between the **try** and **except** keywords) is executed.
- If no exception occurs, the **except clause** is skipped and execution of the **try** statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer **try** statements; if no handler is found, it is an **unhandled exception** and execution stops with a message as shown above.
A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```python
... except (RuntimeError, TypeError, NameError):
    pass
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```python
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the except clauses were reversed (with `except B` first), it
would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```python
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: \{0\}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
raise
```

The `try ... except` statement has an optional `else` clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```python
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
```
The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try ... except statement.

When an exception occurs, it may have an associated value, also known as the exception’s argument. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines __str__() so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```python
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```
If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```python
>>> def this_fails():
...     x = 1/0
... 
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
... 
Handling run-time error: division by zero
```
8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```python
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```python
raise ValueError  # shorthand for 'raise ValueError()
```

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```python
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
... raise

An exception flew by!
```

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
8.5. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the Exception class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```python
class Error(Exception):
    """Base class for exceptions in this module."
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that is not
    allowed."
```
Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter Classes.
8.6. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```python
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
```

A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an except or else clause), it is re-raised after the `finally` clause has been executed. The finally clause is also executed “on the way out” when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example:

```python
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
```
As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.
8.7. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```python
for line in open("myfile.txt"): print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```python
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.
9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object’s members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.
(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)
9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.
9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`), and built-in exception names; the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word attribute for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1]
Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:
- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. To rebind variables found outside of the innermost scope, the nonlocal statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope.
Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

### 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```python
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
```
```python
print("After nonlocal assignment:", spam)
do_global()
print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the `local` assignment (which is default) didn’t change `scope_test`'s binding of `spam`. The `nonlocal` assignment changed `scope_test`'s binding of `spam`, and the `global` assignment changed the module-level binding.

You can also see that there was no previous binding for `spam` before the `global` assignment.
9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1. Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    ...
    ...
    <statement-N>
```

Class definitions, like function definitions (``def`` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an ``if`` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is
created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

9.3.2. Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

```python
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the
class. For example (assuming the above class):

```python
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```python
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```python
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```python
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```
9.3.3. Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

_data attributes_ correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```python
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)

del x.counter
```

The other kind of instance attribute reference is a _method_. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as
MyClass.f — it is a method object, not a function object.

9.3.4. Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print hello world until the end of time.

What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object
before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```python
class Dog:
    kind = 'canine' # class variable shared

    def __init__(self, name):
        self.name = name # instance variable unique

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # shared by all dogs
'canine'
>>> e.kind # shared by all dogs
'canine'
>>> d.name # unique to d
'Fido'
```
As discussed in A Word About Names and Objects, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```python
class Dog:
    tricks = []  # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks          # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```python
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []  # creates a new empty list for each dog
```
```python
def add_trick(self, trick):
    self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```
9.4. Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special
meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```python
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g

Now f, g and h are all attributes of class C that refer to function objects, and consequently they are all methods of instances of C — h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the self argument:

```python
class Bag:
    def __init__(self):
        self.data = []
```
```python
def add(self, x):
    self.data.append(x)

def addtwice(self, x):
    self.add(x)
    self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a class (also called its type). It is stored as `object.__class__`. 
9.5. Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```python
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```python
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.
Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be True only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is True since `bool` is a subclass of `int`. However, `issubclass(float, int)` is False since `float` is not a subclass of `int`.

### 9.5.1. Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```python
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
```
For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see [https://www.python.org/download/releases/2.3/mro/](https://www.python.org/download/releases/2.3/mro/).
9.6. Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

__update = update  # private copy of original update method
```
class MappingSubclass(Mapping):
    
def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)

The above example would work even if MappingSubclass were to introduce a __update identifier since it is replaced with _Mapping__update in the Mapping class and _MappingSubclass__update in the MappingSubclass class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing __dict__ directly.
9.7. Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. An empty class definition will do nicely:

```python
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.
9.8. Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```python
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```python
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
```
Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an \_\_iter\_\_( ) method which returns an object with a \_\_next\_\_( ) method. If the class defines \_\_next\_\_( ), then \_\_iter\_\_( ) can just return self:

```python
class Reverse:
    """Iterator for looping over a sequence backwards"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```python
>>> rev = Reverse('spam')
>>> iter(rev)
<_main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...    print(char)
...    ...
```
9.9. Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```python
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```python
>>> for char in reverse('golf'):
...     print(char)
...     f
  l
  o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state,
when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.
9.10. Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```python
>>> sum(i*i for i in range(10))  # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))  # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 181)}

>>> unique_words = set(word for line in page for word

>>> valedictorian = max((student.gpa, student.name)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Footnotes
Except for one thing. Module objects have a secret read-only attribute called \_\_dict\_\_ which returns the dictionary used to implement the module’s namespace; the name \_\_dict\_\_ is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.
10. Brief Tour of the Standard Library
10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```python
>>> import os
>>> os.getcwd()         # Return the current working directory
'C:\Python37'
>>> os.chdir('/server/accesslogs')  # Change current directory
>>> os.system('mkdir today')       # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```python
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```python
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
```
'install_dir'
10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```python
>>> import glob
>>> glob.glob('*py')
['primes.py', 'random.py', 'quote.py']
```
10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance the following output results from running `python demo.py one two three` at the command line:

```python
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

The `getopt` module processes `sys.argv` using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.
10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```python
>>> sys.stderr.write('Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.
10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```python
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```python
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```
10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

```python
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

```python
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)  # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()  # random float
0.17970987693706186
>>> random.randrange(6)  # random integer chosen from range(6)
4
```

The `statistics` module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```python
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```
The SciPy project <https://scipy.org> has many other modules for numerical computations.
10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib.request` for retrieving data from URLs and `smtplib` for sending mail:

```python
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decoding
...         if 'EST' in line or 'EDT' in line:
...             print(line)
<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
...                 '"""To: jcaesar@example.org
...                 From: soothsayer@example.org
...                 ""
...                 "Beware the Ides of March."
...                 """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)
10.8. Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```python
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the '12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of ")
['12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```
10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` and `tarfile`.

```python
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```
10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The timeit module quickly demonstrates a modest performance advantage:

```python
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to timeit's fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.
10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program’s docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```python
def average(values):
    """Computes the arithmetic mean of a list of numbers."

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()  # automatically validate the embedded tests
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```python
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
```
self.assertEqual(average([20, 30, 70]), 40.0)
self.assertEqual(round(average([1, 5, 7]), 1)
with self.assertRaises(ZeroDivisionError):
    average([])
with self.assertRaises(TypeError):
    average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpc.client` and `xmlrpc.server` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `json` package provides robust support for parsing this popular data interchange format. The `csv` module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the `xml.etree.ElementTree`, `xml.dom` and `xml.sax` packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The `sqlite3` module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.
11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.
11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```python
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious
{"a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```python
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red', ...
...     'yellow'], 'blue']]]
... >>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
       'white',
       ['green', 'red']],
      [['magenta', 'yellow'], 'blue']]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```python
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns a list of strings instead of one big string with the wrapped lines.""
```
The `textwrap` module provides the `fill()` method for wrapping text. The `wrap()` method is similar to `fill()` except that it returns a list of strings instead of a single string with newlines to separate the wrapped lines.

```python
>>> print(textwrap.fill(doc, width=40))
\[...
```

The `locale` module accesses a database of culture-specific data formats. The grouping attribute of `locale`'s format function provides a direct way of formatting numbers with group separators:

```python
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()
# get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s.%*f", (conv['currency_symbol'], conv['frac_digits']), x, grouping=True)
'\[$1,234,567.80\]'
```
11.2. Templating

The string module includes a versatile Template class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by $ with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing $$ creates a single escaped $:

```python
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The substitute() method raises a KeyError when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the safe_substitute() method may be more appropriate — it will leave placeholders unchanged if data is missing:

```python
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ... KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```
Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```python
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):
... Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.
11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```python
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):  # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIIH', data[start:start+14])
    crc32, comp_size, uncomp_size, filenamesize, extr

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size  # skip to the next
```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level threading module can run tasks in background while the main program continues to run:

```python
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground')

background.join()  # Wait for the background task
print('Main program waited until background was done')
```

The principal challenge of multi-threaded applications is coordinating
threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the queue module to feed that thread with requests from other threads. Applications using Queue objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.
11.5. Logging

The `logging` module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```python
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning: config file %s not found',
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: DEBUG, INFO, WARNING, ERROR, and CRITICAL.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.
11.6. Weak References

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The weakref module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```python
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)       # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a # does not create a reference
>>> d['primary']    # fetch the object if it is still alive
10
>>> del a          # remove the one reference
>>> gc.collect()   # run garbage collection
0
>>> d['primary']   # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary'] # entry was automatically removed
```

```text
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary'] # entry was automatically removed
```
File "C:/python37/lib/weakref.py", line 46, in __get__
    o = self.data[key]()

KeyError: 'primary'
11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array()` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```python
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque()` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```python
>>> from collections import deque
>>> d = deque(['task1', 'task2', 'task3'])
>>> d.append('task4')
>>> print('Handling', d.popleft())
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
```
for m in gen_moves(node):
    if is_goal(m):
        return m
unsearched.append(m)

In addition to alternative list implementations, the library also offers other tools such as the bisect module with functions for manipulating sorted lists:

```python
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua')]
```

The heapq module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```python
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new element
>>> sorted_data = [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```
11.8. Decimal Floating Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```python
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating
point:

```python
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
```

```python
>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```python
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```
12. Virtual Environments and Packages
12.1. Introduction

Python applications will often use packages and modules that don’t come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library’s interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A’s environment.
12.2. Creating Virtual Environments

The module used to create and manage virtual environments is called `venv`. `venv` will usually install the most recent version of Python that you have available. If you have multiple versions of Python on your system, you can select a specific Python version by running `python3` or whichever version you want.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python3 -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter, the standard library, and various supporting files.

Once you've created a virtual environment, you may activate it.

On Windows, run:

```
tutorial-env\Scripts\activate.bat
```

On Unix or MacOS, run:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the `csh` or `fish` shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)
Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ..., '~/envs/tutorial-env/lib/python3.5/site-packages']
>>> 
```
12.3. Managing Packages with pip

You can install, upgrade, and remove packages using a program called pip. By default pip will install packages from the Python Package Index, <https://pypi.org>. You can browse the Python Package Index by going to it in your web browser, or you can use pip's limited search feature:

(tutorial-env) $ pip search astronomy
skyfield - Elegant astronomy for Python
gary - Galactic astronomy and gravitational dynamics.
novas - The United States Naval Observatory NOVAS astronomy library
astroobs - Provides astronomy ephemerides
PyAstronomy - A collection of astronomy related tools
...

pip has a number of subcommands: “search”, “install”, “uninstall”, “freeze”, etc. (Consult the Installing Python Modules guide for complete documentation for pip.)

You can install the latest version of a package by specifying a package's name:

(tutorial-env) $ pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3

You can also install a specific version of a package by giving the package name followed by `==` and the version number:
(tutorial-env) $ pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0

If you re-run this command, pip will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run pip install --upgrade to upgrade the package to the latest version:

(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0

pip uninstall followed by one or more package names will remove the packages from the virtual environment.

pip show will display information about a particular package:

(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/pyt
pip list will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

pip freeze will produce a similar list of the installed packages, but the output uses the format that pip install expects. A common convention is to put this list in a requirements.txt file:

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The requirements.txt can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with install -r:

```
(tutorial-env) $ pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt)
... Collecting numpy==1.9.2 (from -r requirements.txt)
... Collecting requests==2.7.0 (from -r requirements.txt)
Installing collected packages: novas, numpy, request
```
Running `setup.py install` for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0

`pip` has many more options. Consult the `Installing Python Modules` guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the `Distributing Python Modules` guide.
13. What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python’s documentation set. Some other documents in the set are:

- **The Python Standard Library**: You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot* of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what’s available.

- **Installing Python Modules** explains how to install additional modules written by other Python users.

- **The Python Language Reference**: A detailed explanation of Python’s syntax and semantics. It’s heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- [https://www.python.org](https://www.python.org): The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror
may be faster than the main site, depending on your geographical location.

- https://docs.python.org: Fast access to Python’s documentation.
- https://pypi.org: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- http://www.pyvideo.org collects links to Python-related videos from conferences and user-group meetings.
- https://scipy.org: The Scientific Python project includes modules for fast array computations and manipulations plus a host of packages for such things as linear algebra, Fourier transforms, non-linear solvers, random number distributions, statistical analysis and the like.

For Python-related questions and problem reports, you can post to the newsgroup comp.lang.python, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are hundreds of postings a day, asking (and answering) questions, suggesting new features, and announcing new modules. Mailing list archives are available at https://mail.python.org/pipermail/.

Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.
14. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports various styles of editing. This library has its own documentation which we won’t duplicate here.
14.1. Tab Completion and History Editing

Completion of variable and module names is automatically enabled at interpreter startup so that the Tab key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `\'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.
14.2. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.
15. Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

0.125

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction

0.001

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

0.3

or, better,
or, better,

0.333

and so on. No matter how many digits you’re willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

In the same way, no matter how many base 2 digits you’re willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

0.0001100110011001100110011001100110011001100110011001100...

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of 1/10, the binary fraction is $3602879701896397 / 2^{55}$ which is close to but not exactly equal to the true value of 1/10.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

>>> 0.1
0.10000000000000000000000000555111512312578270211815834045410

That is more digits than most people find useful, so Python keeps the
number of digits manageable by displaying a rounded value instead

```python
>>> 1 / 10
0.1
```

Just remember, even though the printed result looks like the exact value of 1/10, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers 0.1 and 0.10000000000000001 and 0.10000000000000000555111512312578270211815834045410 are all approximated by 3602879701896397 / 2 ** 55. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, 0.10000000000000001. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display 0.1.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You’ll see the same kind of thing in all languages that support your hardware’s floating-point arithmetic (although some languages may not display the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```python
>>> format(math.pi, '.12g') # give 12 significant digits
```
It's important to realize that this is, in a real sense, an illusion: you're simply rounding the display of the true machine value.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

```python
>>> .1 + .1 + .1 == .3
False
```

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with `round()` function cannot help:

```python
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `round()` function can be useful for post-rounding so that results with inexact values become comparable to one another:

```python
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is explained in precise detail below, in the
“Representation Error” section. See *The Perils of Floating Point* for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don’t be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in $2^{**53}$ per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method’s format specifiers in *Format String Syntax*.

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like 1/3 can be represented exactly).

If you are a heavy user of floating point operations you should take a look at the Numerical Python package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a
float as a fraction:

```python
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```python
>>> x == 3537115888337719 / 1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```python
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```python
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the `math.fsum()` function which helps mitigate loss-of-precision during summation. It tracks “lost digits” as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
15.1. Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

*Representation error* refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect.

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 “double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where $J$ is an integer containing exactly 53 bits.

Rewriting

$$1 / 10 \approx J / (2^N)$$

as

$$J \approx 2^N / 10$$

and recalling that $J$ has exactly 53 bits (is $\geq 2^{52}$ but $< 2^{53}$), the best value for $N$ is 56:

```python
>>> 2**52 <= 2**56 // 10 < 2**53
True
```
That is, 56 is the only value for $N$ that leaves $J$ with exactly 53 bits. The best possible value for $J$ is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in 754 double precision is:

```
7205759403792794 / 2 ** 56
```

Dividing both the numerator and denominator by two reduces the fraction to:

```
3602879701896397 / 2 ** 55
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be exactly 1/10!

So the computer never “sees” 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by $10^{55}$, we can see the value out to 55
decimal digits:

```python
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value

0.1000000000000000055511151231257827021181583404541015625.

Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```python
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The `fractions` and `decimal` modules make these calculations easy:

```python
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```
16. Appendix
16.1. Interactive Mode

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually `Control-C` or `Delete`) to the primary or secondary prompt cancels the input and returns to the primary prompt. [1] Typing an interrupt while a command is executing raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

16.1.2. Executable Python Scripts

On BSD‘ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#!/usr/bin/env python3.5
```

(assuming that the interpreter is on the user’s PATH) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows
The script can be given an executable mode, or permission, using the **chmod** command.

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates .py files with python.exe so that a double-click on a Python file will run it as a script. The extension can also be .pyw, in that case, the console window that normally appears is suppressed.

### 16.1.3. The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named **PYTHONSTARTUP** to the name of a file containing your start-up commands. This is similar to the .profile feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when /dev/tty is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts sys.ps1 and sys.ps2 in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like **if**
os.path.isfile('.pythonrc.py'):
exec(open('.pythonrc.py').read()). If you want to use the
startup file in a script, you must do this explicitly in the script:

import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)

16.1.4. The Customization Modules

Python provides two hooks to let you customize it: sitecustomize
and usercustomize. To see how it works, you need first to find the
location of your user site-packages directory. Start Python and run this
code:

>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'

Now you can create a file named usercustomize.py in that
directory and put anything you want in it. It will affect every invocation
of Python, unless it is started with the -s option to disable the
automatic import.

sitecustomize works in the same way, but is typically created by
an administrator of the computer in the global site-packages directory,
and is imported before usercustomize. See the documentation of
the site module for more details.
A problem with the GNU Readline package may prevent this.
1. Command line and environment

The CPython interpreter scans the command line and the environment for various settings.

**CPython implementation detail:** Other implementations’ command line schemes may differ. See [Alternate Implementations](#) for further resources.
1.1. Command line

When invoking Python, you may specify any of these options:

```
python [-bBdEhiIOqsSuvVWx?] [-c command | -m module-name]
```

The most common use case is, of course, a simple invocation of a script:

```
python myscript.py
```

1.1.1. Interface options

The interpreter interface resembles that of the UNIX shell, but provides some additional methods of invocation:

- When called with standard input connected to a tty device, it prompts for commands and executes them until an EOF (an end-of-file character, you can produce that with `Ctrl-D` on UNIX or `Ctrl-Z`, `Enter` on Windows) is read.
- When called with a file name argument or with a file as standard input, it reads and executes a script from that file.
- When called with a directory name argument, it reads and executes an appropriately named script from that directory.
- When called with `-c command`, it executes the Python statement(s) given as `command`. Here `command` may contain multiple statements separated by newlines. Leading whitespace is significant in Python statements!
- When called with `-m module-name`, the given module is located on the Python module path and executed as a script.

In non-interactive mode, the entire input is parsed before it is
executed.

An interface option terminates the list of options consumed by the interpreter, all consecutive arguments will end up in `sys.argv` – note that the first element, subscript zero (`sys.argv[0]`), is a string reflecting the program's source.

```
-c <command>
```

Execute the Python code in *command*. *command* can be one or more statements separated by newlines, with significant leading whitespace as in normal module code.

If this option is given, the first element of `sys.argv` will be "-c" and the current directory will be added to the start of `sys.path` (allowing modules in that directory to be imported as top level modules).

```
-m <module-name>
```

Search `sys.path` for the named module and execute its contents as the `__main__` module.

Since the argument is a *module* name, you must not give a file extension (.py). The module name should be a valid absolute Python module name, but the implementation may not always enforce this (e.g. it may allow you to use a name that includes a hyphen).

Package names (including namespace packages) are also permitted. When a package name is supplied instead of a normal module, the interpreter will execute `<pkg>.__main__` as the main module. This behaviour is deliberately similar to the handling of directories and zipfiles that are passed to the interpreter as the script argument.
Note: This option cannot be used with built-in modules and extension modules written in C, since they do not have Python module files. However, it can still be used for precompiled modules, even if the original source file is not available.

If this option is given, the first element of `sys.argv` will be the full path to the module file (while the module file is being located, the first element will be set to "-m"). As with the `-c` option, the current directory will be added to the start of `sys.path`.

Many standard library modules contain code that is invoked on their execution as a script. An example is the `timeit` module:

```python
python -mtimeit -s "setup here" 'benchmarked code'
python -mtimeit -h # for details
```

See also:
`runpy.run_module()`
Equivalent functionality directly available to Python code

PEP 338 – Executing modules as scripts

*Changed in version 3.1:* Supply the package name to run a `__main__` submodule.

*Changed in version 3.4:* namespace packages are also supported.

Read commands from standard input (`sys.stdin`). If standard input is a terminal, `-i` is implied.

If this option is given, the first element of `sys.argv` will be "-" and the current directory will be added to the start of `sys.path`. 
Execute the Python code contained in `script`, which must be a filesystem path (absolute or relative) referring to either a Python file, a directory containing a `__main__.py` file, or a zipfile containing a `__main__.py` file.

If this option is given, the first element of `sys.argv` will be the script name as given on the command line.

If the script name refers directly to a Python file, the directory containing that file is added to the start of `sys.path`, and the file is executed as the `__main__` module.

If the script name refers to a directory or zipfile, the script name is added to the start of `sys.path` and the `__main__.py` file in that location is executed as the `__main__` module.

**See also:**

`runpy.run_path()`  
Equivalent functionality directly available to Python code

If no interface option is given, `-i` is implied, `sys.argv[0]` is an empty string ("") and the current directory will be added to the start of `sys.path`. Also, tab-completion and history editing is automatically enabled, if available on your platform (see Readline configuration).

**See also:** Invoking the Interpreter

*Changed in version 3.4:* Automatic enabling of tab-completion and history editing.
1.1.2. Generic options

-?  
-h  
--help
   Print a short description of all command line options.

-V  
--version
   Print the Python version number and exit. Example output could be:

   Python 3.7.0b2+

When given twice, print more information about the build, like:

   Python 3.7.0b2+ (3.7:0c076caaa8, Sep 22 2018, 12: [GCC 6.2.0 20161005]

   New in version 3.6: The -VV option.

1.1.3. Miscellaneous options

-b
   Issue a warning when comparing bytes or bytearray with str or bytes with int. Issue an error when the option is given twice (-bb).

   Changed in version 3.5: Affects comparisons of bytes with int.

-B
If given, Python won't try to write `.pyc` files on the import of source modules. See also `PYTHONDONTWRITEBYTECODE`.

**--check-hash-based-pycs** default|always|never

Control the validation behavior of hash-based `.pyc` files. SeeCached bytecode invalidation. When set to `default`, checked and unchecked hash-based bytecode cache files are validated according to their default semantics. When set to `always`, all hash-based `.pyc` files, whether checked or unchecked, are validated against their corresponding source file. When set to `never`, hash-based `.pyc` files are not validated against their corresponding source files.

The semantics of timestamp-based `.pyc` files are unaffected by this option.

**-d**

Turn on parser debugging output (for expert only, depending on compilation options). See also `PYTHONDEBUG`.

**-E**

Ignore all PYTHON* environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

**-i**

When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal. The `PYTHONSTARTUP` file is not read.

This can be useful to inspect global variables or a stack trace when a script raises an exception. See also `PYTHONINSPECT`. 
-I
Run Python in isolated mode. This also implies -E and -s. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory. All PYTHON* environment variables are ignored, too. Further restrictions may be imposed to prevent the user from injecting malicious code.

*New in version 3.4.*

-0
Remove assert statements and any code conditional on the value of `__debug__`. Augment the filename for compiled (bytecode) files by adding `.opt-1` before the `.pyc` extension (see PEP 488). See also PYTHONOPTIMIZE.

*Changed in version 3.5: Modify `.pyc` filenames according to PEP 488.*

-00
Do -0 and also discard docstrings. Augment the filename for compiled (bytecode) files by adding `.opt-2` before the `.pyc` extension (see PEP 488).

*Changed in version 3.5: Modify `.pyc` filenames according to PEP 488.*

-q
Don't display the copyright and version messages even in interactive mode.

*New in version 3.2.*

-R
Turn on hash randomization. This option only has an effect if the `PYTHONHASHSEED` environment variable is set to 0, since hash randomization is enabled by default.

On previous versions of Python, this option turns on hash randomization, so that the `__hash__()` values of str, bytes and datetime are “salted” with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

Hash randomization is intended to provide protection against a denial-of-service caused by carefully-chosen inputs that exploit the worst case performance of a dict construction, O(n^2) complexity. See http://www.ocert.org/advisories/ocert-2011-003.html for details.

`PYTHONHASHSEED` allows you to set a fixed value for the hash seed secret.

*Changed in version 3.7:* The option is no longer ignored.

*New in version 3.2.3.*

**-s**

Don't add the user site-packages directory to `sys.path`.

**See also:** PEP 370 – Per user site-packages directory

**-S**

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).
-u

Force the stdout and stderr streams to be unbuffered. This option has no effect on the stdin stream.

See also PYTHONUNBUFFERED.

Changed in version 3.7: The text layer of the stdout and stderr streams now is unbuffered.

-v

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. When given twice (-vv), print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit. See also PYTHONVERBOSE.

-W arg

Warning control. Python's warning machinery by default prints warning messages to sys.stderr. A typical warning message has the following form:

file:line: category: message

By default, each warning is printed once for each source line where it occurs. This option controls how often warnings are printed.

Multiple -W options may be given; when a warning matches more than one option, the action for the last matching option is performed. Invalid -W options are ignored (though, a warning message is printed about invalid options when the first warning is issued).

Warnings can also be controlled using the PYTHONWARNINGS
environment variable and from within a Python program using the `warnings` module.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
-Wdefault  # Warn once per call location
-Werror   # Convert to exceptions
-Walways  # Warn every time
-Wmodule  # Warn once per calling module
-Wonce    # Warn once per Python process
-Wignore  # Never warn
```

The action names can be abbreviated as desired (e.g. `-Wi`, `-Wd`, `-Wa`, `-We`) and the interpreter will resolve them to the appropriate action name.

See The Warnings Filter and Describing Warning Filters for more details.

**-x**

Skip the first line of the source, allowing use of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

**-X**

Reserved for various implementation-specific options. CPython currently defines the following possible values:

- `-X faulthandler` to enable `faulthandler`;
- `-X showrefcount` to output the total reference count and number of used memory blocks when the program finishes or after each statement in the interactive interpreter. This only works on debug builds.
-X tracemalloc to start tracing Python memory allocations using the tracemalloc module. By default, only the most recent frame is stored in a traceback of a trace. Use -X tracemalloc=NFRAME to start tracing with a traceback limit of NFRAME frames. See the tracemalloc.start() for more information.

- X showalloccount to output the total count of allocated objects for each type when the program finishes. This only works when Python was built with COUNT_ALLOCS defined.

- X importtime to show how long each import takes. It shows module name, cumulative time (including nested imports) and self time (excluding nested imports). Note that its output may be broken in multi-threaded application. Typical usage is python3 -X importtime -c 'import asyncio'. See also PYTHONPROFILEIMPORTTIME.

- X dev: enable CPython’s “development mode”, introducing additional runtime checks which are too expensive to be enabled by default. It should not be more verbose than the default if the code is correct: new warnings are only emitted when an issue is detected. Effect of the developer mode:
  ◦ Add default warning filter, as -W default.
  ◦ Install debug hooks on memory allocators: see the PyMem_SetupDebugHooks() C function.
  ◦ Enable the faulthandler module to dump the Python traceback on a crash.
  ◦ Enable asyncio debug mode.
  ◦ Set the dev_mode attribute of sys.flags to True.

- X utf8 enables UTF-8 mode for operating system interfaces, overriding the default locale-aware mode. -X utf8=0 explicitly disables UTF-8 mode (even when it would otherwise activate automatically). See PYTHONUTF8 for more details.
It also allows passing arbitrary values and retrieving them through the `sys._xoptions` dictionary.

*Changed in version 3.2:* The `-X` option was added.

*New in version 3.3:* The `-X faulthandler` option.

*New in version 3.4:* The `-X showrefcount` and `-X tracemalloc` options.

*New in version 3.6:* The `-X showallococount` option.

*New in version 3.7:* The `-X importtime`, `-X dev` and `-X utf8` options.

### 1.1.4. Options you shouldn’t use

- `-J`
  
  Reserved for use by Jython.
1.2. Environment variables

These environment variables influence Python’s behavior, they are processed before the command-line switches other than -E or -l. It is customary that command-line switches override environmental variables where there is a conflict.

**PYTHONHOME**

Change the location of the standard Python libraries. By default, the libraries are searched in `prefix/lib/pythonversion` and `exec_prefix/lib/pythonversion`, where `prefix` and `exec_prefix` are installation-dependent directories, both defaulting to `/usr/local`.

When `PYTHONHOME` is set to a single directory, its value replaces both `prefix` and `exec_prefix`. To specify different values for these, set `PYTHONHOME` to `prefix:exec_prefix`.

**PYTHONPATH**

Augment the default search path for module files. The format is the same as the shell's `PATH`: one or more directory pathnames separated by `os.pathsep` (e.g. colons on Unix or semicolons on Windows). Non-existent directories are silently ignored.

In addition to normal directories, individual `PYTHONPATH` entries may refer to zipfiles containing pure Python modules (in either source or compiled form). Extension modules cannot be imported from zipfiles.

The default search path is installation dependent, but generally begins with `prefix/lib/pythonversion` (see `PYTHONHOME`)
above). It is *always* appended to `PYTHONPATH`.

An additional directory will be inserted in the search path in front of `PYTHONPATH` as described above under Interface options. The search path can be manipulated from within a Python program as the variable `sys.path`.

**PYTHONSTARTUP**

If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode. The file is executed in the same namespace where interactive commands are executed so that objects defined or imported in it can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` and the hook `sys.__interactivehook__` in this file.

**PYTHONOPTIMIZE**

If this is set to a non-empty string it is equivalent to specifying the `-O` option. If set to an integer, it is equivalent to specifying `-O multiple times.

**PYTHONBREAKPOINT**

If this is set, it names a callable using dotted-path notation. The module containing the callable will be imported and then the callable will be run by the default implementation of `sys.breakpointhook()` which itself is called by built-in `breakpoint()`. If not set, or set to the empty string, it is equivalent to the value “`pdb.set_trace`”. Setting this to the string “0” causes the default implementation of `sys.breakpointhook()` to do nothing but return immediately.

*New in version 3.7.*
**PYTHONDEBUG**
If this is set to a non-empty string it is equivalent to specifying the `-d` option. If set to an integer, it is equivalent to specifying `-d` multiple times.

**PYTHONINSPECT**
If this is set to a non-empty string it is equivalent to specifying the `-i` option.

This variable can also be modified by Python code using `os.environ` to force inspect mode on program termination.

**PYTHONUNBUFFERED**
If this is set to a non-empty string it is equivalent to specifying the `-u` option.

**PYTHONVERBOSE**
If this is set to a non-empty string it is equivalent to specifying the `-v` option. If set to an integer, it is equivalent to specifying `-v` multiple times.

**PYTHONCASEOK**
If this is set, Python ignores case in `import` statements. This only works on Windows and OS X.

**PYTHONDONTWRITEBYTECODE**
If this is set to a non-empty string, Python won’t try to write `.pyc` files on the import of source modules. This is equivalent to specifying the `-B` option.

**PYTHONHASHSEED**
If this variable is not set or set to `random`, a random value is used
to seed the hashes of str, bytes and datetime objects.

If `PYTHONHASHSEED` is set to an integer value, it is used as a fixed seed for generating the hash() of the types covered by the hash randomization.

Its purpose is to allow repeatable hashing, such as for selftests for the interpreter itself, or to allow a cluster of python processes to share hash values.

The integer must be a decimal number in the range [0,4294967295]. Specifying the value 0 will disable hash randomization.

*New in version 3.2.3.*

**PYTHONIOENCODING**

If this is set before running the interpreter, it overrides the encoding used for stdin/stdout/stderr, in the syntax `encodingname:errorhandler`. Both the `encodingname` and the `:errorhandler` parts are optional and have the same meaning as in `str.encode()`.

For stderr, the `:errorhandler` part is ignored; the handler will always be 'backslashreplace'.

*Changed in version 3.4:* The `encodingname` part is now optional.

*Changed in version 3.6:* On Windows, the encoding specified by this variable is ignored for interactive console buffers unless `PYTHONLEGACYWINDOWSSTDIO` is also specified. Files and pipes redirected through the standard streams are not affected.

**PYTHONNOUSERSITE**
If this is set, Python won’t add the **user site-packages directory** to **sys.path**.

**See also:**  [PEP 370](https://www.python.org/dev/peps/pep-0370/) – Per user site-packages directory

**PYTHONUSERBASE**

Defines the **user base directory**, which is used to compute the path of the **user site-packages directory** and **Distutils installation paths** for `python setup.py install --user`.

**See also:**  [PEP 370](https://www.python.org/dev/peps/pep-0370/) – Per user site-packages directory

**PYTHONEXECUTABLE**

If this environment variable is set, `sys.argv[0]` will be set to its value instead of the value got through the C runtime. Only works on Mac OS X.

**PYTHONWARNINGS**

This is equivalent to the **-W** option. If set to a comma separated string, it is equivalent to specifying **-W** multiple times, with filters later in the list taking precedence over those earlier in the list.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
PYTHONWARNINGS=default  # Warn once per call location
PYTHONWARNINGS=error    # Convert to exceptions
PYTHONWARNINGS=always   # Warn every time
PYTHONWARNINGS=module   # Warn once per calling module
PYTHONWARNINGS=once     # Warn once per Python process
PYTHONWARNINGS=ignore   # Never warn
```
See The Warnings Filter and Describing Warning Filters for more details.

**PYTHONFAULTHANDLER**

If this environment variable is set to a non-empty string, `faulthandler.enable()` is called at startup: install a handler for SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. This is equivalent to `-X faulthandler` option.

*New in version 3.3.*

**PYTHONTRACEMALLOC**

If this environment variable is set to a non-empty string, start tracing Python memory allocations using the `tracemalloc` module. The value of the variable is the maximum number of frames stored in a traceback of a trace. For example, `PYTHONTRACEMALLOC=1` stores only the most recent frame. See the `tracemalloc.start()` for more information.

*New in version 3.4.*

**PYTHONPROFILEIMPORTTIME**

If this environment variable is set to a non-empty string, Python will show how long each import takes. This is exactly equivalent to setting `-X importtime` on the command line.

*New in version 3.7.*

**PYTHONASYNCIODEBUG**

If this environment variable is set to a non-empty string, enable the debug mode of the `asyncio` module.
New in version 3.4.

**PYTHONMALLOC**

Set the Python memory allocators and/or install debug hooks.

Set the family of memory allocators used by Python:

- **default**: use the default memory allocators.
- **malloc**: use the malloc() function of the C library for all domains (PYMEM_DOMAIN_RAW, PYMEM_DOMAIN_MEM, PYMEM_DOMAIN_OBJ).
- **pymalloc**: use the pymalloc allocator for PYMEM_DOMAIN_MEM and PYMEM_DOMAIN_OBJ domains and use the malloc() function for the PYMEM_DOMAIN_RAW domain.

Install debug hooks:

- **debug**: install debug hooks on top of the default memory allocators.
- **malloc_debug**: same as malloc but also install debug hooks
- **pymalloc_debug**: same as pymalloc but also install debug hooks

See the default memory allocators and the `PyMem_SetupDebugHooks()` function (install debug hooks on Python memory allocators).

*Changed in version 3.7*: Added the "default" allocator.

*New in version 3.6.*

**PYTHONMALLOCSTATS**

If set to a non-empty string, Python will print statistics of the
**pymalloc memory allocator**  every time a new pymalloc object arena is created, and on shutdown.

This variable is ignored if the **PYTHONMALLOC** environment variable is used to force the malloc() allocator of the C library, or if Python is configured without pymalloc support.

*Changed in version 3.6:* This variable can now also be used on Python compiled in release mode. It now has no effect if set to an empty string.

**PYTHONLEGACYWINDOWSFSENCODING**

If set to a non-empty string, the default filesystem encoding and errors mode will revert to their pre-3.6 values of ‘mbcs’ and ‘replace’, respectively. Otherwise, the new defaults ‘utf-8’ and ‘surrogatepass’ are used.

This may also be enabled at runtime with `sys._enablelegacywindowsfsencoding()`.

*Availability:* Windows.

*New in version 3.6:* See PEP 529 for more details.

**PYTHONLEGACYWINDOWSSTDIO**

If set to a non-empty string, does not use the new console reader and writer. This means that Unicode characters will be encoded according to the active console code page, rather than using utf-8.

This variable is ignored if the standard streams are redirected (to files or pipes) rather than referring to console buffers.

*Availability:* Windows.
New in version 3.6.

**PYTHONCOERCECLOCALE**

If set to the value 0, causes the main Python command line application to skip coercing the legacy ASCII-based C and POSIX locales to a more capable UTF-8 based alternative.

If this variable is *not* set (or is set to a value other than 0), the LC_ALL locale override environment variable is also not set, and the current locale reported for the LC_CTYPE category is either the default C locale, or else the explicitly ASCII-based POSIX locale, then the Python CLI will attempt to configure the following locales for the LC_CTYPE category in the order listed before loading the interpreter runtime:

- C.UTF-8
- C.utf8
- UTF-8

If setting one of these locale categories succeeds, then the LC_CTYPE environment variable will also be set accordingly in the current process environment before the Python runtime is initialized. This ensures that in addition to being seen by both the interpreter itself and other locale-aware components running in the same process (such as the GNU readline library), the updated setting is also seen in subprocesses (regardless of whether or not those processes are running a Python interpreter), as well as in operations that query the environment rather than the current C locale (such as Python's own `locale.getdefaultlocale()`).

Configuring one of these locales (either explicitly or via the above implicit locale coercion) automatically enables the surrogateescape error handler for `sys.stdin` and
sys.stdout  (sys.stderr continues to use backslashreplace as it does in any other locale). This stream handling behavior can be overridden using PYTHONIOENCODING as usual.

For debugging purposes, setting PYTHONCOERCECLOCALE=warn will cause Python to emit warning messages on stderr if either the locale coercion activates, or else if a locale that would have triggered coercion is still active when the Python runtime is initialized.

Also note that even when locale coercion is disabled, or when it fails to find a suitable target locale, PYTHONUTF8 will still activate by default in legacy ASCII-based locales. Both features must be disabled in order to force the interpreter to use ASCII instead of UTF-8 for system interfaces.

Availability: *nix.

New in version 3.7: See PEP 538 for more details.

PYTHONDDEVMODE

If this environment variable is set to a non-empty string, enable the CPython “development mode”. See the -X dev option.

New in version 3.7.

PYTHONUTF8

If set to 1, enables the interpreter’s UTF-8 mode, where UTF-8 is used as the text encoding for system interfaces, regardless of the current locale setting.

This means that:
- `sys.getfilesystemencoding()` returns 'UTF-8' (the locale encoding is ignored).
- `locale.getpreferredencoding()` returns 'UTF-8' (the locale encoding is ignored, and the function's `do_setlocale` parameter has no effect).
- `sys.stdin`, `sys.stdout`, and `sys.stderr` all use UTF-8 as their text encoding, with the surrogateescape error handler being enabled for `sys.stdin` and `sys.stdout` (`sys.stderr` continues to use backslashreplace as it does in the default locale-aware mode).

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- Command line arguments, environment variables and filenames are decoded to text using the UTF-8 encoding.
- `os.fsdecode()` and `os.fsencode()` use the UTF-8 encoding.
- `open()`, `io.open()`, and `codecs.open()` use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

Note that the standard stream settings in UTF-8 mode can be overridden by `PYTHONIOENCODING` (just as they can be in the default locale-aware mode).

If set to 0, the interpreter runs in its default locale-aware mode.

Setting any other non-empty string causes an error during interpreter initialisation.
If this environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

Also available as the `-X utf8` option.

**Availability:** *nix.

*New in version 3.7:* See PEP 540 for more details.

### 1.2.1. Debug-mode variables

Setting these variables only has an effect in a debug build of Python, that is, if Python was configured with the `--with-pydebug` build option.

**PYTHONTHREADDEBUG**

If set, Python will print threading debug info.

**PYTHONDUMPREFS**

If set, Python will dump objects and reference counts still alive after shutting down the interpreter.
2. Using Python on Unix platforms
2.1. Getting and installing the latest version of Python

2.1.1. On Linux

Python comes preinstalled on most Linux distributions, and is available as a package on all others. However there are certain features you might want to use that are not available on your distro's package. You can easily compile the latest version of Python from source.

In the event that Python doesn't come preinstalled and isn't in the repositories as well, you can easily make packages for your own distro. Have a look at the following links:

See also:

  for Debian users

https://en.opensuse.org/Portal:Packaging
  for OpenSuse users

  for Fedora users

  for Slackware users

2.1.2. On FreeBSD and OpenBSD

- FreeBSD users, to add the package use:
pkg install python3

- OpenBSD users, to add the package use:

```bash
pkg_add -r python
```

For example i386 users get the 2.5.1 version of Python using:

```bash
pkg_add ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/i386/python-2.5.1p2.tgz
```

### 2.1.3. On OpenSolaris

You can get Python from [OpenCSW](https://www.opensolaris.org). Various versions of Python are available and can be installed with e.g. `pkgutil -i python27`. 
2.2. Building Python

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release’s source or just grab a fresh clone. (If you want to contribute patches, you will need a clone.)

The build process consists in the usual

```
./configure
make
make install
```

invocations. Configuration options and caveats for specific Unix platforms are extensively documented in the README.rst file in the root of the Python source tree.

**Warning:** make install can overwrite or masquerade the python3 binary. make altinstall is therefore recommended instead of make install since it only installs `exec_prefix/bin/pythonversion`. 
2.3. Python-related paths and files

These are subject to difference depending on local installation conventions; prefix (${prefix}) and exec_prefix (${exec_prefix}) are installation-dependent and should be interpreted as for GNU software; they may be the same.

For example, on most Linux systems, the default for both is /usr.

<table>
<thead>
<tr>
<th>File/directory</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec_prefix/bin/python3</td>
<td>Recommended location of the interpreter.</td>
</tr>
<tr>
<td>prefix/lib/pythonversion, exec_prefix/lib/pythonversion</td>
<td>Recommended locations of the directories containing the standard modules.</td>
</tr>
<tr>
<td>prefix/include/pythonversion, exec_prefix/include/pythonversion</td>
<td>Recommended locations of the directories containing the include files needed for developing Python extensions and embedding the interpreter.</td>
</tr>
</tbody>
</table>
2.4. Miscellaneous

To easily use Python scripts on Unix, you need to make them executable, e.g. with

```bash
$ chmod +x script
```

and put an appropriate Shebang line at the top of the script. A good choice is usually

```bash
#!/usr/bin/env python3
```

which searches for the Python interpreter in the whole `PATH`. However, some Unices may not have the `env` command, so you may need to hardcode `/usr/bin/python3` as the interpreter path.

To use shell commands in your Python scripts, look at the `subprocess` module.
2.5. Editors and IDEs

There are a number of IDEs that support Python programming language. Many editors and IDEs provide syntax highlighting, debugging tools, and PEP 8 checks.

Please go to Python Editors and Integrated Development Environments for a comprehensive list.
3. Using Python on Windows

This document aims to give an overview of Windows-specific behaviour you should know about when using Python on Microsoft Windows.

Unlike most Unix systems and services, Windows does not include a system supported installation of Python. To make Python available, the CPython team has compiled Windows installers (MSI packages) with every release for many years. These installers are primarily intended to add a per-user installation of Python, with the core interpreter and library being used by a single user. The installer is also able to install for all users of a single machine, and a separate ZIP file is available for application-local distributions.

As specified in PEP 11, a Python release only supports a Windows platform while Microsoft considers the platform under extended support. This means that Python 3.7 supports Windows Vista and newer. If you require Windows XP support then please install Python 3.4.

There are a number of different installers available for Windows, each with certain benefits and downsides.

The full installer contains all components and is the best option for developers using Python for any kind of project.

The Microsoft Store package is a simple installation of Python that is suitable for running scripts and packages, and using IDLE or other development environments. It requires Windows 10, but can be safely installed without corrupting other programs. It also provides many convenient commands for launching Python and its tools.
The nuget.org packages are lightweight installations intended for continuous integration systems. It can be used to build Python packages or run scripts, but is not updateable and has no user interface tools.

The embeddable package is a minimal package of Python suitable for embedding into a larger application.
3.1. The full installer

3.1.1. Installation steps

Four Python 3.7 installers are available for download - two each for the 32-bit and 64-bit versions of the interpreter. The web installer is a small initial download, and it will automatically download the required components as necessary. The offline installer includes the components necessary for a default installation and only requires an internet connection for optional features. See Installing Without Downloading for other ways to avoid downloading during installation.

After starting the installer, one of two options may be selected:

If you select “Install Now”:

- You will not need to be an administrator (unless a system update for the C Runtime Library is required or you install the Python
Launcher for Windows for all users)

- Python will be installed into your user directory
- The Python Launcher for Windows will be installed according to the option at the bottom of the first page
- The standard library, test suite, launcher and pip will be installed
- If selected, the install directory will be added to your PATH
- Shortcuts will only be visible for the current user

Selecting “Customize installation” will allow you to select the features to install, the installation location and other options or post-install actions. To install debugging symbols or binaries, you will need to use this option.

To perform an all-users installation, you should select “Customize installation”. In this case:

- You may be required to provide administrative credentials or approval
- Python will be installed into the Program Files directory
- The Python Launcher for Windows will be installed into the Windows directory
- Optional features may be selected during installation
- The standard library can be pre-compiled to bytecode
- If selected, the install directory will be added to the system PATH
- Shortcuts are available for all users

3.1.2. Removing the MAX_PATH Limitation

Windows historically has limited path lengths to 260 characters. This meant that paths longer than this would not resolve and errors would result.

In the latest versions of Windows, this limitation can be expanded to approximately 32,000 characters. Your administrator will need to
activate the “Enable Win32 long paths” group policy, or set the registry value
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem@LongPathsEnabled
to 1.

This allows the open() function, the os module and most other path functionality to accept and return paths longer than 260 characters when using strings. (Use of bytes as paths is deprecated on Windows, and this feature is not available when using bytes.)

After changing the above option, no further configuration is required.

Changed in version 3.6: Support for long paths was enabled in Python.

3.1.3. Installing Without UI

All of the options available in the installer UI can also be specified from the command line, allowing scripted installers to replicate an installation on many machines without user interaction. These options may also be set without suppressing the UI in order to change some of the defaults.

To completely hide the installer UI and install Python silently, pass the /quiet option. To skip past the user interaction but still display progress and errors, pass the /passive option. The /uninstall option may be passed to immediately begin removing Python - no prompt will be displayed.

All other options are passed as name=value, where the value is usually 0 to disable a feature, 1 to enable a feature, or a path. The full list of available options is shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstallAllUsers</td>
<td>Perform a system-wide installation.</td>
<td>0</td>
</tr>
<tr>
<td>TargetDir</td>
<td>The installation directory</td>
<td>Selected based on InstallAllUsers</td>
</tr>
<tr>
<td>DefaultAllUsersTargetDir</td>
<td>The default installation directory for all-user installs</td>
<td>%ProgramFiles%\Python</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%ProgramFiles(x86%</td>
</tr>
<tr>
<td>DefaultJustForMeTargetDir</td>
<td>The default installation directory for just-for-me installs</td>
<td>%LocalAppData%\Programs\Python🤔</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%LocalAppData%\Programs\Python-32</td>
</tr>
<tr>
<td>DefaultCustomTargetDir</td>
<td>The default custom install directory displayed in the UI</td>
<td>(empty)</td>
</tr>
<tr>
<td>AssociateFiles</td>
<td>Create file associations if the launcher is also installed.</td>
<td>1</td>
</tr>
<tr>
<td>CompileAll</td>
<td>Compile all .py files to .pyc.</td>
<td>0</td>
</tr>
<tr>
<td>PrependPath</td>
<td>Add install and Scripts directories to PATH and .PY to</td>
<td>0</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Shortcuts</td>
<td>Create shortcuts for the interpreter, documentation and IDLE if installed.</td>
<td>1</td>
</tr>
<tr>
<td>Include_doc</td>
<td>Install Python manual</td>
<td>1</td>
</tr>
<tr>
<td>Include_debug</td>
<td>Install debug binaries</td>
<td>0</td>
</tr>
<tr>
<td>Include_dev</td>
<td>Install developer headers and libraries</td>
<td>1</td>
</tr>
<tr>
<td>Include_exe</td>
<td>Install <code>python.exe</code> and related files</td>
<td>1</td>
</tr>
<tr>
<td>Include_launcher</td>
<td>Install Python Launcher for Windows.</td>
<td>1</td>
</tr>
<tr>
<td>InstallLauncherAllUsers</td>
<td>Installs Python Launcher for Windows for all users.</td>
<td>1</td>
</tr>
<tr>
<td>Include_lib</td>
<td>Install standard library and extension modules</td>
<td>1</td>
</tr>
<tr>
<td>Install bundled</td>
<td>Install bundled</td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Include_pip</td>
<td>pip and setuptools</td>
<td>1</td>
</tr>
<tr>
<td>Include_symbols</td>
<td>Install debugging symbols (*.pdb)</td>
<td>0</td>
</tr>
<tr>
<td>Include_tcltk</td>
<td>Install Tcl/Tk support and IDLE</td>
<td>1</td>
</tr>
<tr>
<td>Include_test</td>
<td>Install standard library test suite</td>
<td>1</td>
</tr>
<tr>
<td>Include_tools</td>
<td>Install utility scripts</td>
<td>1</td>
</tr>
<tr>
<td>LauncherOnly</td>
<td>Only installs the launcher. This will override most other options.</td>
<td>0</td>
</tr>
<tr>
<td>SimpleInstall</td>
<td>Disable most install UI</td>
<td>0</td>
</tr>
<tr>
<td>SimpleInstallDescription</td>
<td>A custom message to display when the simplified install UI is used.</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

For example, to silently install a default, system-wide Python installation, you could use the following command (from an elevated command prompt):
To allow users to easily install a personal copy of Python without the test suite, you could provide a shortcut with the following command. This will display a simplified initial page and disallow customization:

```
python-3.7.0.exe /quiet InstallAllUsers=1 PrependPath=1 Include_test=0
```

(Note that omitting the launcher also omits file associations, and is only recommended for per-user installs when there is also a system-wide installation that included the launcher.)

The options listed above can also be provided in a file named `unattend.xml` alongside the executable. This file specifies a list of options and values. When a value is provided as an attribute, it will be converted to a number if possible. Values provided as element text are always left as strings. This example file sets the same options as the previous example:

```
<Options>
  <Option Name="InstallAllUsers" Value="no" />
  <Option Name="Include_launcher" Value="0" />
  <Option Name="Include_test" Value="no" />
  <Option Name="SimpleInstall" Value="yes" />
  <Option Name="SimpleInstallDescription">Just for me, no test suite.</Option>
</Options>
```

3.1.4. Installing Without Downloading

As some features of Python are not included in the initial installer,
download, selecting those features may require an internet connection. To avoid this need, all possible components may be downloaded on-demand to create a complete layout that will no longer require an internet connection regardless of the selected features. Note that this download may be bigger than required, but where a large number of installations are going to be performed it is very useful to have a locally cached copy.

Execute the following command from Command Prompt to download all possible required files. Remember to substitute `python-3.7.0.exe` for the actual name of your installer, and to create layouts in their own directories to avoid collisions between files with the same name.

```
python-3.7.0.exe /layout [optional target directory]
```

You may also specify the `/quiet` option to hide the progress display.

### 3.1.5. Modifying an install

Once Python has been installed, you can add or remove features through the Programs and Features tool that is part of Windows. Select the Python entry and choose “Uninstall/Change” to open the installer in maintenance mode.

“Modify” allows you to add or remove features by modifying the checkboxes - unchanged checkboxes will not install or remove anything. Some options cannot be changed in this mode, such as the install directory; to modify these, you will need to remove and then reinstall Python completely.

“Repair” will verify all the files that should be installed using the current
settings and replace any that have been removed or modified.

“Uninstall” will remove Python entirely, with the exception of the Python Launcher for Windows, which has its own entry in Programs and Features.
3.2. The Microsoft Store package

*New in version 3.7.2.*

**Note:** The Microsoft Store package is currently considered unstable while its interactions with other tools and other copies of Python are evaluated. While Python itself is stable, this installation method may change its behavior and capabilities during Python 3.7 releases.

The Microsoft Store package is an easily installable Python interpreter that is intended mainly for interactive use, for example, by students.

To install the package, ensure you have the latest Windows 10 updates and search the Microsoft Store app for “Python 3.7”. Ensure that the app you select is published by the Python Software Foundation, and install it.

**Warning:** Python will always be available for free on the Microsoft Store. If you are asked to pay for it, you have not selected the correct package.

After installation, Python may be launched by finding it in Start. Alternatively, it will be available from any Command Prompt or PowerShell session by typing `python`. Further, `pip` and IDLE may be used by typing `pip` or `idle`. IDLE can also be found in Start.

All three commands are also available with version number suffixes, for example, as `python3.exe` and `python3.x.exe` as well as `python.exe` (where `3.x` is the specific version you want to launch, such as 3.7).
Virtual environments can be created with `python -m venv` and activated and used as normal.

If you have installed another version of Python and added it to your PATH variable, it will be available as `python.exe` rather than the one from the Microsoft Store. To access the new installation, use `python3.exe` or `python3.x.exe`.

To remove Python, open Settings and use Apps and Features, or else find Python in Start and right-click to select Uninstall. Uninstalling will remove all packages you installed directly into this Python installation, but will not remove any virtual environments

### 3.2.1. Known Issues

Currently, the `py.exe` launcher cannot be used to start Python when it has been installed from the Microsoft Store.

Because of restrictions on Microsoft Store apps, Python scripts may not have full write access to shared locations such as TEMP and the registry. Instead, it will write to a private copy. If your scripts must modify the shared locations, you will need to install the full installer.
3.3. The nuget.org packages

*New in version 3.5.2.*

The nuget.org package is a reduced size Python environment intended for use on continuous integration and build systems that do not have a system-wide install of Python. While nuget is “the package manager for .NET”, it also works perfectly fine for packages containing build-time tools.

Visit [nuget.org](http://nuget.org) for the most up-to-date information on using nuget. What follows is a summary that is sufficient for Python developers.

The `nuget.exe` command line tool may be downloaded directly from [https://aka.ms/nugetclidl](https://aka.ms/nugetclidl), for example, using `curl` or `PowerShell`. With the tool, the latest version of Python for 64-bit or 32-bit machines is installed using:

```
```

```
```

To select a particular version, add a `-Version 3.x.y`. The output directory may be changed from `.`, and the package will be installed into a subdirectory. By default, the subdirectory is named the same as the package, and without the `-ExcludeVersion` option this name will include the specific version installed. Inside the subdirectory is a `tools` directory that contains the Python installation:

```
# Without -ExcludeVersion
> .\python.3.5.2\tools\python.exe -V
Python 3.5.2
```
In general, nuget packages are not upgradeable, and newer versions should be installed side-by-side and referenced using the full path. Alternatively, delete the package directory manually and install it again. Many CI systems will do this automatically if they do not preserve files between builds.

Alongside the tools directory is a build\native directory. This contains a MSBuild properties file python.props that can be used in a C++ project to reference the Python install. Including the settings will automatically use the headers and import libraries in your build.

The package information pages on nuget.org are www.nuget.org/packages/python for the 64-bit version and www.nuget.org/packages/pythonx86 for the 32-bit version.
3.4. The embeddable package

*New in version 3.5.*

The embedded distribution is a ZIP file containing a minimal Python environment. It is intended for acting as part of another application, rather than being directly accessed by end-users.

When extracted, the embedded distribution is (almost) fully isolated from the user’s system, including environment variables, system registry settings, and installed packages. The standard library is included as pre-compiled and optimized `.pyc` files in a ZIP, and `python3.dll`, `python37.dll`, `python.exe` and `pythonw.exe` are all provided. Tcl/tk (including all dependants, such as Idle), pip and the Python documentation are not included.

**Note:** The embedded distribution does not include the Microsoft C Runtime and it is the responsibility of the application installer to provide this. The runtime may have already been installed on a user’s system previously or automatically via Windows Update, and can be detected by finding `ucrtbase.dll` in the system directory.

Third-party packages should be installed by the application installer alongside the embedded distribution. Using pip to manage dependencies as for a regular Python installation is not supported with this distribution, though with some care it may be possible to include and use pip for automatic updates. In general, third-party packages should be treated as part of the application (“vendoring”) so that the developer can ensure compatibility with newer versions before providing updates to users.
The two recommended use cases for this distribution are described below.

3.4.1. Python Application

An application written in Python does not necessarily require users to be aware of that fact. The embedded distribution may be used in this case to include a private version of Python in an install package. Depending on how transparent it should be (or conversely, how professional it should appear), there are two options.

Using a specialized executable as a launcher requires some coding, but provides the most transparent experience for users. With a customized launcher, there are no obvious indications that the program is running on Python: icons can be customized, company and version information can be specified, and file associations behave properly. In most cases, a custom launcher should simply be able to call `Py_Main` with a hard-coded command line.

The simpler approach is to provide a batch file or generated shortcut that directly calls the `python.exe` or `pythonw.exe` with the required command-line arguments. In this case, the application will appear to be Python and not its actual name, and users may have trouble distinguishing it from other running Python processes or file associations.

With the latter approach, packages should be installed as directories alongside the Python executable to ensure they are available on the path. With the specialized launcher, packages can be located in other locations as there is an opportunity to specify the search path before launching the application.
3.4.2. Embedding Python

Applications written in native code often require some form of scripting language, and the embedded Python distribution can be used for this purpose. In general, the majority of the application is in native code, and some part will either invoke python.exe or directly use python3.dll. For either case, extracting the embedded distribution to a subdirectory of the application installation is sufficient to provide a loadable Python interpreter.

As with the application use, packages can be installed to any location as there is an opportunity to specify search paths before initializing the interpreter. Otherwise, there is no fundamental differences between using the embedded distribution and a regular installation.
3.5. Alternative bundles

Besides the standard CPython distribution, there are modified packages including additional functionality. The following is a list of popular versions and their key features:

ActivePython
   Installer with multi-platform compatibility, documentation, PyWin32

Anaconda
   Popular scientific modules (such as numpy, scipy and pandas) and the conda package manager.

Canopy
   A “comprehensive Python analysis environment” with editors and other development tools.

WinPython
   Windows-specific distribution with prebuilt scientific packages and tools for building packages.

Note that these packages may not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.
3.6. Configuring Python

To run Python conveniently from a command prompt, you might consider changing some default environment variables in Windows. While the installer provides an option to configure the PATH and PATHEXT variables for you, this is only reliable for a single, system-wide installation. If you regularly use multiple versions of Python, consider using the Python Launcher for Windows.

3.6.1. Excursus: Setting environment variables

Windows allows environment variables to be configured permanently at both the User level and the System level, or temporarily in a command prompt.

To temporarily set environment variables, open Command Prompt and use the `set` command:

```
C:\>set PATH=C:\Program Files\Python 3.7;%PATH%
C:\>set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib
C:\>python
```

These changes will apply to any further commands executed in that console, and will be inherited by any applications started from the console.

Including the variable name within percent signs will expand to the existing value, allowing you to add your new value at either the start or the end. Modifying PATH by adding the directory containing `python.exe` to the start is a common way to ensure the correct version of Python is launched.
To permanently modify the default environment variables, click Start and search for ‘edit environment variables’, or open System properties, Advanced system settings and click the Environment Variables button. In this dialog, you can add or modify User and System variables. To change System variables, you need non-restricted access to your machine (i.e. Administrator rights).

**Note:** Windows will concatenate User variables after System variables, which may cause unexpected results when modifying PATH.

The `PYTHONPATH` variable is used by all versions of Python 2 and Python 3, so you should not permanently configure this variable unless it only includes code that is compatible with all of your installed Python versions.

**See also:**

  Environment variables in Windows NT
  The SET command, for temporarily modifying environment variables
  The SETX command, for permanently modifying environment variables
  How To Manage Environment Variables in Windows XP
- [https://www.chem.gla.ac.uk/~louis/software/faq/q1.html](https://www.chem.gla.ac.uk/~louis/software/faq/q1.html)
  Setting Environment variables, Louis J. Farrugia
3.6.2. Finding the Python executable

*Changed in version 3.5.*

Besides using the automatically created start menu entry for the Python interpreter, you might want to start Python in the command prompt. The installer has an option to set that up for you.

On the first page of the installer, an option labelled “Add Python to PATH” may be selected to have the installer add the install location into the `PATH`. The location of the `Scripts\` folder is also added. This allows you to type `python` to run the interpreter, and `pip` for the package installer. Thus, you can also execute your scripts with command line options, see Command line documentation.

If you don’t enable this option at install time, you can always re-run the installer, select Modify, and enable it. Alternatively, you can manually modify the PATH using the directions in Excursus: Setting environment variables. You need to set your `PATH` environment variable to include the directory of your Python installation, delimited by a semicolon from other entries. An example variable could look like this (assuming the first two entries already existed):

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Python
```
3.7. Python Launcher for Windows

New in version 3.3.

The Python launcher for Windows is a utility which aids in locating and executing of different Python versions. It allows scripts (or the command-line) to indicate a preference for a specific Python version, and will locate and execute that version.

Unlike the PATH variable, the launcher will correctly select the most appropriate version of Python. It will prefer per-user installations over system-wide ones, and orders by language version rather than using the most recently installed version.

The launcher was originally specified in PEP 397.

3.7.1. Getting started

3.7.1.1. From the command-line

Changed in version 3.6.

System-wide installations of Python 3.3 and later will put the launcher on your PATH. The launcher is compatible with all available versions of Python, so it does not matter which version is installed. To check that the launcher is available, execute the following command in Command Prompt:

```
py
```

You should find that the latest version of Python you have installed is started - it can be exited as normal, and any additional command-line
arguments specified will be sent directly to Python.

If you have multiple versions of Python installed (e.g., 2.7 and 3.7) you will have noticed that Python 3.7 was started - to launch Python 2.7, try the command:

```
py -2.7
```

If you want the latest version of Python 2.x you have installed, try the command:

```
py -2
```

You should find the latest version of Python 2.x starts.

If you see the following error, you do not have the launcher installed:

```
'py' is not recognized as an internal or external command, operable program or batch file.
```

Per-user installations of Python do not add the launcher to PATH unless the option was selected on installation.

3.7.1.2. Virtual environments

*New in version 3.5.*

If the launcher is run with no explicit Python version specification, and a virtual environment (created with the standard library `venv` module or the external `virtualenv` tool) active, the launcher will run the virtual environment’s interpreter rather than the global one. To run the global interpreter, either deactivate the virtual environment, or explicitly specify the global Python version.
3.7.1.3. From a script

Let's create a test Python script - create a file called `hello.py` with the following contents:

```python
#!/python
import sys
sys.stdout.write("hello from Python %s
" % (sys.version))
```

From the directory in which `hello.py` lives, execute the command:

```
p y hello.py
```

You should notice the version number of your latest Python 2.x installation is printed. Now try changing the first line to be:

```python
#!/python3
```

Re-executing the command should now print the latest Python 3.x information. As with the above command-line examples, you can specify a more explicit version qualifier. Assuming you have Python 2.6 installed, try changing the first line to `#!/python2.6` and you should find the 2.6 version information printed.

Note that unlike interactive use, a bare “python” will use the latest version of Python 2.x that you have installed. This is for backward compatibility and for compatibility with Unix, where the command `python` typically refers to Python 2.

3.7.1.4. From file associations

The launcher should have been associated with Python files (i.e. `.py`,
.pyw, .pyc files) when it was installed. This means that when you double-click on one of these files from Windows explorer the launcher will be used, and therefore you can use the same facilities described above to have the script specify the version which should be used.

The key benefit of this is that a single launcher can support multiple Python versions at the same time depending on the contents of the first line.

### 3.7.2. Shebang Lines

If the first line of a script file starts with `#!`, it is known as a “shebang” line. Linux and other Unix like operating systems have native support for such lines and they are commonly used on such systems to indicate how a script should be executed. This launcher allows the same facilities to be used with Python scripts on Windows and the examples above demonstrate their use.

To allow shebang lines in Python scripts to be portable between Unix and Windows, this launcher supports a number of ‘virtual’ commands to specify which interpreter to use. The supported virtual commands are:

- `/usr/bin/env python`
- `/usr/bin/python`
- `/usr/local/bin/python`
- `python`

For example, if the first line of your script starts with

```bash
#! /usr/bin/python
```

The default Python will be located and used. As many Python scripts
written to work on Unix will already have this line, you should find these scripts can be used by the launcher without modification. If you are writing a new script on Windows which you hope will be useful on Unix, you should use one of the shebang lines starting with /usr.

Any of the above virtual commands can be suffixed with an explicit version (either just the major version, or the major and minor version) - for example /usr/bin/python2.7 - which will cause that specific version to be located and used.

The /usr/bin/env form of shebang line has one further special property. Before looking for installed Python interpreters, this form will search the executable PATH for a Python executable. This corresponds to the behaviour of the Unix env program, which performs a PATH search.

3.7.3. Arguments in shebang lines

The shebang lines can also specify additional options to be passed to the Python interpreter. For example, if you have a shebang line:

```
#! /usr/bin/python -v
```

Then Python will be started with the -v option

3.7.4. Customization

3.7.4.1. Customization via INI files

Two .ini files will be searched by the launcher - py.ini in the current user’s “application data” directory (i.e. the directory returned by calling the Windows function SHGetFolderPath with
CSIDL_LOCAL_APPDATA) and py.ini in the same directory as the launcher. The same .ini files are used for both the ‘console’ version of the launcher (i.e. py.exe) and for the ‘windows’ version (i.e. pyw.exe).

Customization specified in the “application directory” will have precedence over the one next to the executable, so a user, who may not have write access to the .ini file next to the launcher, can override commands in that global .ini file.

3.7.4.2. Customizing default Python versions

In some cases, a version qualifier can be included in a command to dictate which version of Python will be used by the command. A version qualifier starts with a major version number and can optionally be followed by a period (‘.’) and a minor version specifier. If the minor qualifier is specified, it may optionally be followed by “-32” to indicate the 32-bit implementation of that version be used.

For example, a shebang line of #!python has no version qualifier, while #!python3 has a version qualifier which specifies only a major version.

If no version qualifiers are found in a command, the environment variable PY_PYTHON can be set to specify the default version qualifier - the default value is “2”. Note this value could specify just a major version (e.g. “2”) or a major.minor qualifier (e.g. “2.6”), or even major.minor-32.

If no minor version qualifiers are found, the environment variable PY_PYTHON{major} (where {major} is the current major version qualifier as determined above) can be set to specify the full version. If no such option is found, the launcher will enumerate the installed Python versions and use the latest minor release found for the major
version, which is likely, although not guaranteed, to be the most recently installed version in that family.

On 64-bit Windows with both 32-bit and 64-bit implementations of the same (major.minor) Python version installed, the 64-bit version will always be preferred. This will be true for both 32-bit and 64-bit implementations of the launcher - a 32-bit launcher will prefer to execute a 64-bit Python installation of the specified version if available. This is so the behavior of the launcher can be predicted knowing only what versions are installed on the PC and without regard to the order in which they were installed (i.e., without knowing whether a 32 or 64-bit version of Python and corresponding launcher was installed last). As noted above, an optional “-32” suffix can be used on a version specifier to change this behaviour.

Examples:

- If no relevant options are set, the commands `python` and `python2` will use the latest Python 2.x version installed and the command `python3` will use the latest Python 3.x installed.
- The commands `python3.1` and `python2.7` will not consult any options at all as the versions are fully specified.
- If `PY_PYTHON=3`, the commands `python` and `python3` will both use the latest installed Python 3 version.
- If `PY_PYTHON=3.1-32`, the command `python` will use the 32-bit implementation of 3.1 whereas the command `python3` will use the latest installed Python (PY_PYTHON was not considered at all as a major version was specified.)
- If `PY_PYTHON=3` and `PY_PYTHON3=3.1`, the commands `python` and `python3` will both use specifically 3.1

In addition to environment variables, the same settings can be configured in the .INI file used by the launcher. The section in the INI
file is called [defaults] and the key name will be the same as the environment variables without the leading PY_ prefix (and note that the key names in the INI file are case insensitive.) The contents of an environment variable will override things specified in the INI file.

For example:

- Setting PY_PYTHON=3.1 is equivalent to the INI file containing:

```
[defaults]
python=3.1
```

- Setting PY_PYTHON=3 and PY_PYTHON3=3.1 is equivalent to the INI file containing:

```
[defaults]
python=3
python3=3.1
```

### 3.7.5. Diagnostics

If an environment variable PYLAUNCH_DEBUG is set (to any value), the launcher will print diagnostic information to stderr (i.e. to the console). While this information manages to be simultaneously verbose and terse, it should allow you to see what versions of Python were located, why a particular version was chosen and the exact command-line used to execute the target Python.
3.8. Finding modules

Python usually stores its library (and thereby your site-packages folder) in the installation directory. So, if you had installed Python to C:\Python\, the default library would reside in C:\Python\Lib\ and third-party modules should be stored in C:\Python\Lib\site-packages\.

To completely override sys.path, create a .pth file with the same name as the DLL (python37._pth) or the executable (python._pth) and specify one line for each path to add to sys.path. The file based on the DLL name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and site is not imported unless one line in the file specifies import site. Blank paths and lines starting with # are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to site are not permitted, and arbitrary code cannot be specified.

Note that .pth files (without leading underscore) will be processed normally by the site module when import site has been specified.

When no .pth file is found, this is how sys.path is populated on Windows:

- An empty entry is added at the start, which corresponds to the current directory.
- If the environment variable `PYTHONPATH` exists, as described in Environment variables, its entries are added next. Note that on Windows, paths in this variable must be separated by semicolons, to distinguish them from the colon used in drive identifiers (C: \ etc.).
- Additional “application paths” can be added in the registry as subkeys of `\SOFTWARE\Python\PythonCore{version}\PythonPath` under both the `HKEY_CURRENT_USER` and `HKEY_LOCAL_MACHINE` hives. Subkeys which have semicolon-delimited path strings as their default value will cause each path to be added to `sys.path`. (Note that all known installers only use HKLM, so HKCU is typically empty.)
- If the environment variable `PYTHONHOME` is set, it is assumed as “Python Home”. Otherwise, the path of the main Python executable is used to locate a “landmark file” (either `Lib\os.py` or `pythonXY.zip`) to deduce the “Python Home”. If a Python home is found, the relevant sub-directories added to `sys.path` (Lib, plat-win, etc) are based on that folder. Otherwise, the core Python path is constructed from the PythonPath stored in the registry.
- If the Python Home cannot be located, no `PYTHONPATH` is specified in the environment, and no registry entries can be found, a default path with relative entries is used (e.g. `.\Lib; .\plat-win`, etc).

If a `pyvenv.cfg` file is found alongside the main executable or in the directory one level above the executable, the following variations apply:

- If home is an absolute path and `PYTHONHOME` is not set, this path is used instead of the path to the main executable when deducing the home location.
The end result of all this is:

- When running python.exe, or any other .exe in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is deduced, and the core paths in the registry are ignored. Other “application paths” in the registry are always read.
- When Python is hosted in another .exe (different directory, embedded via COM, etc), the “Python Home” will not be deduced, so the core path from the registry is used. Other “application paths” in the registry are always read.
- If Python can’t find its home and there are no registry value (frozen .exe, some very strange installation setup) you get a path with some default, but relative, paths.

For those who want to bundle Python into their application or distribution, the following advice will prevent conflicts with other installations:

- Include a .pth file alongside your executable containing the directories to include. This will ignore paths listed in the registry and environment variables, and also ignore site unless import site is listed.
- If you are loading python3.dll or python37.dll in your own executable, explicitly call Py_SetPath() or (at least) Py_SetProgramName() before Py_Initialize().
- Clear and/or overwrite PYTHONPATH and set PYTHONHOME before launching python.exe from your application.
- If you cannot use the previous suggestions (for example, you are a distribution that allows people to run python.exe directly), ensure that the landmark file (Lib\os.py) exists in your install directory. (Note that it will not be detected inside a ZIP file, but a correctly named ZIP file will be detected instead.)
These will ensure that the files in a system-wide installation will not take precedence over the copy of the standard library bundled with your application. Otherwise, your users may experience problems using your application. Note that the first suggestion is the best, as the others may still be susceptible to non-standard paths in the registry and user site-packages.

*Changed in version 3.6:*

- Adds `.pth` file support and removes `applocal` option from `pyvenv.cfg`.
- Adds `pythonXX.zip` as a potential landmark when directly adjacent to the executable.

*Deprecated since version 3.6:*

Modules specified in the registry under `Modules` (not `PythonPath`) may be imported by `importlib.machinery.WindowsRegistryFinder`. This finder is enabled on Windows in 3.6.0 and earlier, but may need to be explicitly added to `sys.meta_path` in the future.
3.9. Additional modules

Even though Python aims to be portable among all platforms, there are features that are unique to Windows. A couple of modules, both in the standard library and external, and snippets exist to use these features.

The Windows-specific standard modules are documented in MS Windows Specific Services.

3.9.1. PyWin32

The PyWin32 module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

- Component Object Model (COM)
- Win32 API calls
- Registry
- Event log
- Microsoft Foundation Classes (MFC) user interfaces

PythonWin is a sample MFC application shipped with PyWin32. It is an embeddable IDE with a built-in debugger.

See also:

Win32 How Do I...?
by Tim Golden

Python and COM
by David and Paul Boddie
3.9.2. cx_Freeze

cx_Freeze is a distutils extension (see Extending Distutils) which wraps Python scripts into executable Windows programs (*.exe files). When you have done this, you can distribute your application without requiring your users to install Python.

3.9.3. WConio

Since Python’s advanced terminal handling layer, curses, is restricted to Unix-like systems, there is a library exclusive to Windows as well: Windows Console I/O for Python.

WConio is a wrapper for Turbo-C’s CONIO.H, used to create text user interfaces.
3.10. Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release’s source or just grab a fresh checkout.

The source tree contains a build solution and project files for Microsoft Visual Studio 2015, which is the compiler used to build the official Python releases. These files are in the PCbuild directory.

Check PCbuild/readme.txt for general information on the build process.

For extension modules, consult Building C and C++ Extensions on Windows.

See also:

Python + Windows + distutils + SWIG + gcc MinGW
or “Creating Python extensions in C/C++ with SWIG and compiling them with MinGW gcc under Windows” or “Installing Python extension with distutils and without Microsoft Visual C++” by Sébastien Sauvage, 2003

MingW – Python extensions
by Trent Apter et al, 2007
3.11. Other Platforms

With ongoing development of Python, some platforms that used to be supported earlier are no longer supported (due to the lack of users or developers). Check PEP 11 for details on all unsupported platforms.

- **Windows CE** is still supported.
- The Cygwin installer offers to install the Python interpreter as well (cf. Cygwin package source, Maintainer releases)

See Python for Windows for detailed information about platforms with pre-compiled installers.
4. Using Python on a Macintosh

Author: Bob Savage <bobsavage@mac.com>

Python on a Macintosh running Mac OS X is in principle very similar to Python on any other Unix platform, but there are a number of additional features such as the IDE and the Package Manager that are worth pointing out.
4.1. Getting and Installing MacPython

Mac OS X 10.8 comes with Python 2.7 pre-installed by Apple. If you wish, you are invited to install the most recent version of Python 3 from the Python website (https://www.python.org). A current “universal binary” build of Python, which runs natively on the Mac’s new Intel and legacy PPC CPU’s, is available there.

What you get after installing is a number of things:

- A MacPython 3.6 folder in your Applications folder. In here you find IDLE, the development environment that is a standard part of official Python distributions; PythonLauncher, which handles double-clicking Python scripts from the Finder; and the “Build Applet” tool, which allows you to package Python scripts as standalone applications on your system.
- A framework /Library/Frameworks/Python.framework, which includes the Python executable and libraries. The installer adds this location to your shell path. To uninstall MacPython, you can simply remove these three things. A symlink to the Python executable is placed in /usr/local/bin/.

The Apple-provided build of Python is installed in /System/Library/Frameworks/Python.framework and /usr/bin/python, respectively. You should never modify or delete these, as they are Apple-controlled and are used by Apple- or third-party software. Remember that if you choose to install a newer Python version from python.org, you will have two different but functional Python installations on your computer, so it will be important that your paths and usages are consistent with what you want to do.

IDLE includes a help menu that allows you to access Python
documentation. If you are completely new to Python you should start reading the tutorial introduction in that document.

If you are familiar with Python on other Unix platforms you should read the section on running Python scripts from the Unix shell.

4.1.1. How to run a Python script

Your best way to get started with Python on Mac OS X is through the IDLE integrated development environment, see section The IDE and use the Help menu when the IDE is running.

If you want to run Python scripts from the Terminal window command line or from the Finder you first need an editor to create your script. Mac OS X comes with a number of standard Unix command line editors, vim and emacs among them. If you want a more Mac-like editor, BBEdit or TextWrangler from Bare Bones Software (see http://www.barebones.com/products/bbedit/index.html) are good choices, as is TextMate (see https://macromates.com/). Other editors include Gvim (http://macvim-dev.github.io/macvim/) and Aquamacs (http://aquamacs.org/).

To run your script from the Terminal window you must make sure that /usr/local/bin is in your shell search path.

To run your script from the Finder you have two options:

- Drag it to PythonLauncher
- Select PythonLauncher as the default application to open your script (or any .py script) through the finder Info window and double-click it. PythonLauncher has various preferences to control how your script is launched. Option-dragging allows you to change these for one invocation, or use its Preferences menu to
change things globally.

4.1.2. Running scripts with a GUI

With older versions of Python, there is one Mac OS X quirk that you need to be aware of: programs that talk to the Aqua window manager (in other words, anything that has a GUI) need to be run in a special way. Use `pythonw` instead of `python` to start such scripts.

With Python 3.6, you can use either `python` or `pythonw`.

4.1.3. Configuration

Python on OS X honors all standard Unix environment variables such as `PYTHONPATH`, but setting these variables for programs started from the Finder is non-standard as the Finder does not read your `.profile` or `.cshrc` at startup. You need to create a file `~/.MacOSX/environment.plist`. See Apple’s Technical Document QA1067 for details.

For more information on installation Python packages in MacPython, see section Installing Additional Python Packages.
4.2. The IDE

MacPython ships with the standard IDLE development environment. A good introduction to using IDLE can be found at http://www.hashcollision.org/hkn/python/idle_intro/index.html.
4.3. Installing Additional Python Packages

There are several methods to install additional Python packages:

- Packages can be installed via the standard Python distutils mode (`python setup.py install`).
- Many packages can also be installed via the `setuptools` extension or `pip` wrapper, see https://pip.pypa.io/.
4.4. GUI Programming on the Mac

There are several options for building GUI applications on the Mac with Python.

*PyObjC* is a Python binding to Apple’s Objective-C/Cocoa framework, which is the foundation of most modern Mac development. Information on PyObjC is available from [https://pythonhosted.org/pyobjc/](https://pythonhosted.org/pyobjc/).

The standard Python GUI toolkit is *tkinter*, based on the cross-platform Tk toolkit ([https://www.tcl.tk](https://www.tcl.tk)). An Aqua-native version of Tk is bundled with OS X by Apple, and the latest version can be downloaded and installed from [https://www.activestate.com](https://www.activestate.com); it can also be built from source.

*wxPython* is another popular cross-platform GUI toolkit that runs natively on Mac OS X. Packages and documentation are available from [https://www.wxpython.org](https://www.wxpython.org).

*PyQt* is another popular cross-platform GUI toolkit that runs natively on Mac OS X. More information can be found at [https://riverbankcomputing.com/software/pyqt/intro](https://riverbankcomputing.com/software/pyqt/intro).
4.5. Distributing Python Applications on the Mac

The “Build Applet” tool that is placed in the MacPython 3.6 folder is fine for packaging small Python scripts on your own machine to run as a standard Mac application. This tool, however, is not robust enough to distribute Python applications to other users.

The standard tool for deploying standalone Python applications on the Mac is **py2app**. More information on installing and using py2app can be found at [http://undefined.org/python/#py2app](http://undefined.org/python/#py2app).
4.6. Other Resources

The MacPython mailing list is an excellent support resource for Python users and developers on the Mac:

https://www.python.org/community/sigs/current/pythonmac-sig/

Another useful resource is the MacPython wiki:

https://wiki.python.org/moin/MacPython
1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, CPython is the one Python implementation in widespread use (although alternate implementations continue to gain support), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you’ll find short “implementation notes” sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are documented in The Python Standard Library. A few built-in modules are mentioned when they interact in a significant way with the language definition.
1.1. Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython
This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

Jython
Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

Python for .NET
This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. It was created by Brian Lloyd. For more information, see the Python for .NET home page.

IronPython
An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

PyPy
An implementation of Python written completely in Python. It supports several advanced features not found in other implementations like stackless support and a Just in Time compiler.
One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project’s home page.

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what’s covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you’re using.
1.2. Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

```plaintext
name ::= lc_letter (lc_letter | "_")*
lc_letter ::= "a"..."z"
```

The first line says that a name is an lc_letter followed by a sequence of zero or more lc_letters and underscores. An lc_letter in turn is any of the single characters 'a' through 'z'. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and ::= . A vertical bar (|) is used to separate alternatives; it is the least binding operator in this notation. A star (*) means zero or more repetitions of the preceding item; likewise, a plus (+) means one or more repetitions, and a phrase enclosed in square brackets ([ ]) means zero or one occurrences (in other words, the enclosed phrase is optional). The * and + operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (<>...) gives an
informal description of the symbol defined; e.g., this could be used to describe the notion of ‘control character’ if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter (“Lexical Analysis”) are lexical definitions; uses in subsequent chapters are syntactic definitions.
2. Lexical analysis

A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8, see PEP 3120 for details. If the source file cannot be decoded, a SyntaxError is raised.
2.1. Line structure

A Python program is divided into a number of logical lines.

2.1.1. Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more physical lines by following the explicit or implicit line joining rules.

2.1.2. Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the \n character, representing ASCII LF, is the line terminator).

2.1.3. Comments

A comment starts with a hash character (#) that is not part of a string
literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens.

2.1.4. Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

```
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar’s VIM.

If no encoding declaration is found, the default encoding is UTF-8. In addition, if the first bytes of the file are the UTF-8 byte-order mark (b'\xef\xbb\xbf'), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft’s notepad).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, including string literals, comments and identifiers.

2.1.5. Explicit line joining
Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```python
if 1900 < year < 2100 and 1 <= month <= 12 \
    and 1 <= day <= 31 and 0 <= hour < 24 \
    and 0 <= minute < 60 and 0 <= second < 60:  # Looks like a valid date
    return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

### 2.1.6. Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```python
month_names = ['Januari', 'Februari', 'Maart', 'April', 'Mei', 'Juni', 'Juli', 'Augustus', 'September', 'Oktober', 'November', 'December']
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted
strings (see below); in that case they cannot carry comments.

2.1.7. Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.8. Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line’s indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a TabError is raised in that case.

**Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces
and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line’s indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```python
def perm(l):
    # Compute the list of all permutations of l
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        r.append(s)
    return r
```
The following example shows various indentation errors:

```python
def perm(l):
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of `return r` does not match a level popped off the stack.)

### 2.1.9. Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., `ab` is one token, but `a b` is two tokens).
2.2. Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.
2.3. Identifiers and keywords

Identifiers (also referred to as names) are described by the following lexical definitions.

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also PEP 3131 for further details.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the same as in Python 2.x: the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9.

Python 3.0 introduces additional characters from outside the ASCII range (see PEP 3131). For these characters, the classification uses the version of the Unicode Character Database as included in the unicodedata module.

Identifiers are unlimited in length. Case is significant.

```
identifier ::= xid_start xid_continue*
id_start ::= <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characters with the Other_ID_Start property>
id_continue ::= <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and others with the Other_ID_Continue property>
xid_start ::= <all characters in id_start whose NFKC normalization is in id_start xid_continue*>xid_continue ::= <all characters in id_continue whose NFKC normalization is in id_continue*>
```

The Unicode category codes mentioned above stand for:

- \textit{L} u - uppercase letters
- \textit{L} l - lowercase letters
- \textit{L} t - titlecase letters
- \textit{L} m - modifier letters
• *Lo* - other letters
• *NI* - letter numbers
• *Mn* - nonspaceing marks
• *Mc* - spacing combining marks
• *Nd* - decimal numbers
• *Pc* - connector punctuations
• *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
• *Other_ID_Continue* - likewise

All identifiers are converted into the normal form NFKC while parsing; comparison of identifiers is based on NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 4.1 can be found at https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html.

### 2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

<table>
<thead>
<tr>
<th>False</th>
<th>await</th>
<th>else</th>
<th>import</th>
<th>pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>break</td>
<td>except</td>
<td>in</td>
<td>raise</td>
</tr>
<tr>
<td>True</td>
<td>class</td>
<td>finally</td>
<td>is</td>
<td>return</td>
</tr>
<tr>
<td>and</td>
<td>continue</td>
<td>for</td>
<td>lambda</td>
<td>try</td>
</tr>
<tr>
<td>as</td>
<td>def</td>
<td>from</td>
<td>nonlocal</td>
<td>while</td>
</tr>
<tr>
<td>assert</td>
<td>del</td>
<td>global</td>
<td>not</td>
<td>with</td>
</tr>
<tr>
<td>async</td>
<td>elif</td>
<td>if</td>
<td>or</td>
<td>yield</td>
</tr>
</tbody>
</table>

### 2.3.2. Reserved classes of identifiers
Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

_ *

Not imported by from module import *. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the builtins module. When not in interactive mode, _ has no special meaning and is not defined. See section The import statement.

Note: The name _ is often used in conjunction with internationalization; refer to the documentation for the gettext module for more information on this convention.

__*__

System-defined names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the Special method names section and elsewhere. More will likely be defined in future versions of Python. Any use of __*__ names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

__*

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between “private” attributes of base and derived classes. See section Identifiers (Names).
2.4. Literals

Literals are notations for constant values of some built-in types.

2.4.1. String and Bytes literals

String literals are described by the following lexical definitions:

```
stringliteral ::= [stringprefix](shortstring | longstring)
stringprefix ::= "r" | "u" | "R" | "U" | "f" | ",fr" | "Fr" | "fR" | "FR" | "fr" | "rF" | "Rf" | "RF"
shortstring ::= "'" shortstringitem* "'" | "'
longstring ::= "'''" longstringitem* "'''" | 
shortstringitem ::= shortstringchar | stringescapeseq
longstringitem ::= longstringchar | stringescapeseq
shortstringchar ::= <any source character except \
longstringchar ::= <any source character except \
stringescapeseq ::= "\\" <any source character>
```

```
bytesliteral ::= bytesprefix(shortbytes | longbytes)
bytesprefix ::= "b" | "B" | "br" | "Br" | "bR"
shortbytes ::= "'" shortbytesitem* "'" | "'
longbytes ::= "'''" longbytesitem* "'''" | 
shortbytesitem ::= shortbyteschar | bytesescapeseq
longbytesitem ::= longbyteschar | bytesescapeseq
shortbyteschar ::= <any ASCII character except \
longbyteschar ::= <any ASCII character except \
bytesescapeseq ::= "\\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the stringprefix or bytesprefix and the rest of the literal. The source character set is
defined by the encoding declaration; it is UTF-8 if no encoding declaration is given in the source file; see section Encoding declarations.

In plain English: Both types of literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

Bytes literals are always prefixed with 'b' or 'B'; they produce an instance of the bytes type instead of the str type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escapes.

Both string and bytes literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called *raw strings* and treat backslashes as literal characters. As a result, in string literals, '\u' and '\U' escapes in raw strings are not treated specially. Given that Python 2.x's raw unicode literals behave differently than Python 3.x's the 'ur' syntax is not supported.

*New in version 3.3:* The 'rb' prefix of raw bytes literals has been added as a synonym of 'br'.

*New in version 3.3:* Support for the unicode legacy literal (u'value') was reintroduced to simplify the maintenance of dual Python 2.x and 3.x codebases. See [PEP 414](https://www.python.org/dev/peps/pep-0414/) for more information.

A string literal with 'f' or 'F' in its prefix is a *formatted string literal*; see [Formatted string literals](https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals). The 'f' may be combined with 'r', but
not with 'b' or 'u', therefore raw formatted strings are possible, but formatted bytes literals are not.

In triple-quoted literals, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal. (A “quote” is the character used to open the literal, i.e. either ' or ".)

Unless an 'r' or 'R' prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\newline</td>
<td>Backslash and newline ignored</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>Backslash ()</td>
<td></td>
</tr>
<tr>
<td>'</td>
<td>Single quote (')</td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote (&quot;)</td>
<td></td>
</tr>
<tr>
<td>\a</td>
<td>ASCII Bell (BEL)</td>
<td></td>
</tr>
<tr>
<td>\b</td>
<td>ASCII Backspace (BS)</td>
<td></td>
</tr>
<tr>
<td>\f</td>
<td>ASCII Formfeed (FF)</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>ASCII Linefeed (LF)</td>
<td></td>
</tr>
<tr>
<td>\r</td>
<td>ASCII Carriage Return (CR)</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>ASCII Horizontal Tab (TAB)</td>
<td></td>
</tr>
<tr>
<td>\v</td>
<td>ASCII Vertical Tab (VT)</td>
<td></td>
</tr>
<tr>
<td>\ooo</td>
<td>Character with octal value ooo</td>
<td>(1,3)</td>
</tr>
<tr>
<td>\xhh</td>
<td>Character with hex value hh</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>
Escape sequences only recognized in string literals are:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\N{name}</td>
<td>Character named name in the Unicode database</td>
<td>(4)</td>
</tr>
<tr>
<td>\uxxxx</td>
<td>Character with 16-bit hex value xxxx</td>
<td>(5)</td>
</tr>
<tr>
<td>\Uxxxxxxxxxxx</td>
<td>Character with 32-bit hex value xxxxxxxx</td>
<td>(6)</td>
</tr>
</tbody>
</table>

Notes:

1. As in Standard C, up to three octal digits are accepted.

2. Unlike in Standard C, exactly two hex digits are required.

3. In a bytes literal, hexadecimal and octal escapes denote the byte with the given value. In a string literal, these escapes denote a Unicode character with the given value.

4. Changed in version 3.3: Support for name aliases [1] has been added.

5. Exactly four hex digits are required.

6. Any Unicode character can be encoded this way. Exactly eight hex digits are required.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., the backslash is left in the result. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences only recognized in string
literals fall into the category of unrecognized escapes for bytes literals.

*Changed in version 3.6:* Unrecognized escape sequences produce a DeprecationWarning. In some future version of Python they will be a SyntaxError.

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result; for example, \" in is a valid string literal consisting of two characters: a backslash and a double quote; \" is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, a raw literal cannot end in a single backslash (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the literal, not as a line continuation.

2.4.2. String literal concatenation

Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, "hello" 'world' is equivalent to "helloworld". This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

```python
re.compile("[A-Za-z_]" # letter or underscore
          "[A-Za-z0-9_]" # letter, digit or underscore
          )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The ‘+’ operator must be used to
concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings), and formatted string literals may be concatenated with plain string literals.

2.4.3. Formatted string literals

New in version 3.6.

A formatted string literal or f-string is a string literal that is prefixed with 'f' or 'F'. These strings may contain replacement fields, which are expressions delimited by curly braces {}. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

Escape sequences are decoded like in ordinary string literals (except when a literal is also marked as a raw string). After decoding, the grammar for the contents of the string is:

```
f_string ::= (literal_char | "{{" | "})" replacement_field ::= "{" f_expression ["!" conversion f_expression ::= (conditional_expression | "*") ("," conditional_expression | yield_expression conversion ::= "s" | "r" | "a" format_spec ::= (literal_char | NULL | replacement_field) literal_char ::= <any code point except "{", "}>
```

The parts of the string outside curly braces are treated literally, except that any doubled curly braces `"{{" or "}}" are replaced with the corresponding single curly brace. A single opening curly bracket `"{" marks a replacement field, which starts with a Python expression.
After the expression, there may be a conversion field, introduced by an exclamation point `!' and a colon `:' and a closing curly bracket `'}' A format specifier may also be appended, introduced by a colon `:' and a closing curly bracket `'}' A replacement field ends with a closing curly bracket `'}'.

Expressions in formatted string literals are treated like regular Python expressions surrounded by parentheses, with a few exceptions. An empty expression is not allowed, and a lambda expression must be surrounded by explicit parentheses. Replacement expressions can contain line breaks (e.g. in triple-quoted strings), but they cannot contain comments. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right.

If a conversion is specified, the result of evaluating the expression is converted before formatting. Conversion `!'s' calls `str()` on the result, `!'r' calls `repr()` and `!'a' calls `ascii()`.

The result is then formatted using the `format()` protocol. The format specifier is passed to the `__format__()` method of the expression or conversion result. An empty string is passed when the format specifier is omitted. The formatted result is then included in the final value of the whole string.

Top-level format specifiers may include nested replacement fields. These nested fields may include their own conversion fields and format specifiers, but may not include more deeply-nested replacement fields. The format specifier mini-language is the same as that used by the string `.format()` method.

Formatted string literals may be concatenated, but replacement fields cannot be split across literals.

Some examples of formatted string literals:
A consequence of sharing the same syntax as regular string literals is that characters in the replacement fields must not conflict with the quoting used in the outer formatted string literal:

```python
f"abc {a["x"]} def"  # error: outer string literal
f"abc {a[\'x\']} def"  # workaround: use different quoting
```

Backslashes are not allowed in format expressions and will raise an error:

```python
f"newline: {ord('\n')}"  # raises SyntaxError
```

To include a value in which a backslash escape is required, create a temporary variable:

```python
>>> newline = ord('\n')
>>> f"newline: {newline}"
```
Formatted string literals cannot be used as docstrings, even if they do not include expressions.

```python
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

See also [PEP 498](https://www.python.org/dev/peps/pep-0498/) for the proposal that added formatted string literals, and `str.format()`, which uses a related format string mechanism.

### 2.4.4. Numeric literals

There are three types of numeric literals: integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator `-' and the literal `1`.

### 2.4.5. Integer literals

Integer literals are described by the following lexical definitions:

```
integer ::= decinteger | bininteger | octinteger
decinteger ::= nonzerodigit (["_"] digit)* | "0"-
bininteger ::= "0" ("b" | "B") (["_"] bindigit)+
octinteger ::= "0" ("o" | "0") (["_"] octdigit)+
```
There is no limit for the length of integer literals apart from what can be stored in available memory.

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like 0x.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Some examples of integer literals:

```
7    2147483647    0o177    0b1
3    79228162514264337593543950336    0o377    0xc
      100_000_000_000    0b_1110_0101
```

*Changed in version 3.6:* Underscores are now allowed for grouping purposes in literals.

### 2.4.6. Floating point literals

Floating point literals are described by the following lexical definitions:

```
floatnumber ::= pointfloat | exponentfloat
```
Note that the integer and exponent parts are always interpreted using radix 10. For example, 077e010 is legal, and denotes the same number as 77e10. The allowed range of floating point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Some examples of floating point literals:

```
3.14   10.   .001   1e100   3.14e-10   0e0   3
```

*Changed in version 3.6:* Underscores are now allowed for grouping purposes in literals.

### 2.4.7. Imaginary literals

Imaginary literals are described by the following lexical definitions:

```
imagnumber ::= (floatnumber | digitpart) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., (3+4j). Some examples of imaginary literals:
| 3.14j | 10.j | 10j | .001j | 1e100j | 3.14e-10j |
2.5. Operators

The following tokens are operators:

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>**</th>
<th>/</th>
<th>//</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>==</td>
<td>!=</td>
<td>~</td>
</tr>
</tbody>
</table>
2.6. Delimiters

The following tokens serve as delimiters in the grammar:

| (   ) | [   ] | {   } |
| :   | .   | ;   | @   | =   | ->  |
| +=  | -=  | *=  | /=  | //=  | %=  | @=  |
&=  | |=  | ^=  | >>=  | <<=  | **= |

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis literal. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

| '   | "   | #   | \   |

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

| $   | ?   | `   |

Footnotes

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt
3. Data model
3.1. Objects, values and types

*Objects* are Python’s abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann’s model of a “stored program computer,” code is also represented by objects.)

Every object has an identity, a type and a value. An object’s *identity* never changes once it has been created; you may think of it as the object’s address in memory. The ‘*is*’ operator compares the identity of two objects; the *id()* function returns an integer representing its identity.

**CPython implementation detail:** For CPython, *id(x)* is the memory address where x is stored.

An object’s type determines the operations that the object supports (e.g., “does it have a length?”) and also defines the possible values for objects of that type. The *type()* function returns an object’s type (which is an object itself). Like its identity, an object’s *type* is also unchangeable. [1]

The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter’s value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object’s mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are
mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

**CPython implementation detail:** CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (so you should always close files explicitly).

Note that the use of the implementation’s tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a `try...except` statement may keep objects alive.

Some objects contain references to “external” resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The `try...finally` statement and the `with` statement provide convenient ways to do this.
Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container’s value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after \(a = 1; \ b = 1\), \(a\) and \(b\) may or may not refer to the same object with the value one, depending on the implementation, but after \(c = []; \ d = []\), \(c\) and \(d\) are guaranteed to refer to two different, unique, newly created empty lists. (Note that \(c = d = []\) assigns the same object to both \(c\) and \(d\).)
3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.), although such additions will often be provided via the standard library instead.

Some of the type descriptions below contain a paragraph listing ‘special attributes.’ These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

None
This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don’t explicitly return anything. Its truth value is false.

NotImplemented
This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

See Implementing the arithmetic operations for more details.
Ellipsis

This type has a single value. There is a single object with this value. This object is accessed through the literal ..., or the built-in name Ellipsis. Its truth value is true.

numbers.Number

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

Python distinguishes between integers, floating point numbers, and complex numbers:

numbers.Integral

These represent elements from the mathematical set of integers (positive and negative).

There are two types of integers:

Integers (int)

These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

Booleans (bool)

These represent the truth values False and True. The two objects representing the values False and True are the
only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers.

**numbers.Real** *(float)*

These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

**numbers.Complex** *(complex)*

These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number $z$ can be retrieved through the read-only attributes $z.real$ and $z.imag$.

**Sequences**

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is $n$, the index set
contains the numbers 0, 1, ..., n-1. Item i of sequence a is selected by a[i].

Sequences also support slicing: a[i:j] selects all items with index k such that i <= k < j. When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

Some sequences also support “extended slicing” with a third “step” parameter: a[i:j:k] selects all items of a with index x where x = i + n*k, n >= 0 and i <= x < j.

Sequences are distinguished according to their mutability:

Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings

A string is a sequence of values that represent Unicode code points. All the code points in the range U+0000 - U+10FFFF can be represented in a string. Python doesn't have a char type; instead, every code point in the string is represented as a string object with length 1. The built-in function ord() converts a code point from its string form to an integer in the range 0 - 10FFFF; chr() converts an integer in the range 0 - 10FFFF to the corresponding
length 1 string object. \texttt{str.encode()} can be used to convert a \texttt{str} to \texttt{bytes} using the given text encoding, and \texttt{bytes.decode()} can be used to achieve the opposite.

Tuples

The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a ‘singleton’) can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

Bytes

A bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range $0 \leq x < 256$. Bytes literals (like \texttt{b'abc'}) and the built-in \texttt{bytes()} constructor can be used to create bytes objects. Also, bytes objects can be decoded to strings via the \texttt{decode()} method.

Mutable sequences

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and \texttt{del} (delete) statements.

There are currently two intrinsic mutable sequence types:

Lists

The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)
Byte Arrays

A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable `bytes` objects.

The extension module `array` provides an additional example of a mutable sequence type, as does the `collections` module.

Set types

These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript. However, they can be iterated over, and the built-in function `len()` returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`), only one of them can be contained in a set.

There are currently two intrinsic set types:

Sets

These represent a mutable set. They are created by the built-in `set()` constructor and can be modified afterwards by several methods, such as `add()`.
Frozen sets
These represent an immutable set. They are created by the built-in `frozenset()` constructor. As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key.

Mappings
These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

Dictionaries
These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key’s hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are mutable; they can be created by the `{...}` notation (see section Dictionary displays).

The extension modules `dbm.ndbm` and `dbm.gnu` provide additional examples of mapping types, as does the `collections` module.
Callable types

These are the types to which the function call operation (see section Calls) can be applied:

User-defined functions

A user-defined function object is created by a function definition (see section Function definitions). It should be called with an argument list containing the same number of items as the function’s formal parameter list.

Special attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Writable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>doc</strong></td>
<td>The function’s documentation string, or None if unavailable; not inherited by subclasses</td>
<td>Writable</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The function's name</td>
<td>Writable</td>
</tr>
<tr>
<td><strong>qualname</strong></td>
<td>The function's qualified name</td>
<td>Writable</td>
</tr>
<tr>
<td></td>
<td><em>New in version 3.3.</em></td>
<td></td>
</tr>
<tr>
<td><strong>module</strong></td>
<td>The name of the module the function was defined in, or None if unavailable.</td>
<td>Writable</td>
</tr>
<tr>
<td><strong>defaults</strong></td>
<td>A tuple containing default argument values for those arguments that have defaults, or None if no arguments have a default value</td>
<td>Writable</td>
</tr>
<tr>
<td><strong><strong>code</strong></strong></td>
<td>The code object representing the compiled function body.</td>
<td><strong>Writable</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong><strong>globals</strong></strong></td>
<td>A reference to the dictionary that holds the function's global variables — the global namespace of the module in which the function was defined.</td>
<td><strong>Read-only</strong></td>
</tr>
<tr>
<td><strong><strong>dict</strong></strong></td>
<td>The namespace supporting arbitrary function attributes.</td>
<td><strong>Writable</strong></td>
</tr>
<tr>
<td><strong><strong>closure</strong></strong></td>
<td>None or a tuple of cells that contain bindings for the function's free variables. See below for information on the <code>cell_contents</code> attribute.</td>
<td><strong>Read-only</strong></td>
</tr>
<tr>
<td><strong><strong>annotations</strong></strong></td>
<td>A dict containing annotations of parameters. The keys of the dict are the parameter names, and 'return' for the return annotation, if provided.</td>
<td><strong>Writable</strong></td>
</tr>
<tr>
<td><strong><strong>kwdefaults</strong></strong></td>
<td>A dict containing defaults for keyword-only parameters.</td>
<td><strong>Writable</strong></td>
</tr>
</tbody>
</table>

Most of the attributes labelled “Writable” check the type of the assigned value.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata.
to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

A cell object has the attribute `cell_contents`. This can be used to get the value of the cell, as well as set the value.

Additional information about a function’s definition can be retrieved from its code object; see the description of internal types below.

Instance methods

An instance method object combines a class, a class instance and any callable object (normally a user-defined function).

Special read-only attributes: `__self__` is the class instance object, `__func__` is the function object; `__doc__` is the method’s documentation (same as `__func__.__doc__`); `__name__` is the method name (same as `__func__.__name__`); `__module__` is the name of the module the method was defined in, or None if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object or a class method object.

When an instance method object is created by retrieving a user-defined function object from a class via one of its instances, its
___self___ attribute is the instance, and the method object is said to be bound. The new method’s ___func___ attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the ___func___ attribute of the new instance is not the original method object but its ___func___ attribute.

When an instance method object is created by retrieving a class method object from a class or instance, its ___self___ attribute is the class itself, and its ___func___ attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (___func___) is called, inserting the class instance (___self___) in front of the argument list. For instance, when C is a class which contains a definition for a function f(), and x is an instance of C, calling x.f(1) is equivalent to calling C.f(x, 1).

When an instance method object is derived from a class method object, the “class instance” stored in ___self___ will actually be the class itself, so that calling either x.f(1) or C.f(1) is equivalent to calling f(C, 1) where f is the underlying function.

Note that the transformation from function object to instance method object happens each time the attribute is retrieved from the instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also
notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this only happens when the function is an attribute of the class.

Generator functions
A function or method which uses the `yield` statement (see section The `yield` statement) is called a generator function. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator’s `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

Coroutine functions
A function or method which is defined using `async def` is called a coroutine function. Such a function, when called, returns a coroutine object. It may contain `await` expressions, as well as `async with` and `async for` statements. See also the Coroutine Objects section.

Asynchronous generator functions
A function or method which is defined using `async def` and which uses the `yield` statement is called a asynchronous generator function. Such a function, when called, returns an asynchronous iterator object which can be used in an `async for` statement to execute the body of the function.
Calling the asynchronous iterator’s `__anext__()` method will return an `awaitable` which when awaited will execute until it provides a value using the `yield` expression. When the function executes an empty `return` statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

Built-in functions
A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function's documentation string, or `None` if unavailable; `__name__` is the function's name; `__self__` is set to `None` (but see the next item); `__module__` is the name of the module the function was defined in or `None` if unavailable.

Built-in methods
This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming `alist` is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by `alist`.

Classes
Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override `__new__()`. The arguments of the
call are passed to __new__() and, in the typical case, to
__init__() to initialize the new instance.
Class Instances
Instances of arbitrary classes can be made callable by defining
a __call__() method in their class.
Modules
Modules are a basic organizational unit of Python code, and are
created by the import system as invoked either by the import
statement,
or
by
calling
functions
such
as
importlib.import_module() and built-in __import__(). A
module object has a namespace implemented by a dictionary
object (this is the dictionary referenced by the __globals__
attribute of functions defined in the module). Attribute references
are translated to lookups in this dictionary, e.g., m.x is equivalent
to m.__dict__["x"]. A module object does not contain the code
object used to initialize the module (since it isn’t needed once the
initialization is done).
Attribute assignment updates the module’s namespace dictionary,
e.g., m.x = 1 is equivalent to m.__dict__["x"] = 1.
Predefined (writable) attributes: __name__ is the module’s name;
__doc__ is the module’s documentation string, or None if
unavailable; __annotations__ (optional) is a dictionary
containing variable annotations collected during module body
execution; __file__ is the pathname of the file from which the
module was loaded, if it was loaded from a file. The __file__
attribute may be missing for certain types of modules, such as C
modules that are statically linked into the interpreter; for extension
modules loaded dynamically from a shared library, it is the
pathname of the shared library file.


Special read-only attribute: `__dict__` is the module’s namespace as a dictionary object.

**CPython implementation detail:** Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

Custom classes

Custom class types are typically created by class definitions (see section **Class definitions**). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of ‘diamond’ inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class C, say) would yield a class method object, it is transformed into an instance method object whose `__self__` attribute is C. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section **Implementing Descriptors** for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__`. 
Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

Special attributes: \_\_name\_ is the class name; \_\_module\_ is the module name in which the class was defined; \_\_dict\_ is the dictionary containing the class's namespace; \_\_bases\_ is a tuple containing the base classes, in the order of their occurrence in the base class list; \_\_doc\_ is the class's documentation string, or None if undefined; \_\_annotations\_ (optional) is a dictionary containing variable annotations collected during class body execution.

Class instances
A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose \_\_self\_ attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section Implementing Descriptors for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's \_\_dict\_. If no class attribute is found, and the object's class has a \_\_getattr\_\_(\_) method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's
dictionary, never a class’s dictionary. If the class has a
__setattr__() or __delattr__() method, this is called
instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or
mappings if they have methods with certain special names. See
section Special method names.

Special attributes: __dict__ is the attribute dictionary;
__class__ is the instance’s class.

I/O objects (also known as file objects)
A file object represents an open file. Various shortcuts are available
to create file objects: the open() built-in function, and also
os.popen(), os.fdopen(), and the makefile() method of
socket objects (and perhaps by other functions or methods
provided by extension modules).

The objects sys.stdin, sys.stdout and sys.stderr are
initialized to file objects corresponding to the interpreter’s standard
input, output and error streams; they are all open in text mode and
therefore follow the interface defined by the io.TextIOBase
abstract class.

Internal types
A few types used internally by the interpreter are exposed to the
user. Their definitions may change with future versions of the
interpreter, but they are mentioned here for completeness.

Code objects
Code objects represent byte-compiled executable Python code,
or bytecode. The difference between a code object and a
function object is that the function object contains an explicit
reference to the function’s globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

Special read-only attributes: co_name gives the function name; co_argcount is the number of positional arguments (including arguments with default values); co_nlocals is the number of local variables used by the function (including arguments); co_varnames is a tuple containing the names of the local variables (starting with the argument names); co_cellvars is a tuple containing the names of local variables that are referenced by nested functions; co_freevars is a tuple containing the names of free variables; co_code is a string representing the sequence of bytecode instructions; co_consts is a tuple containing the literals used by the bytecode; co_names is a tuple containing the names used by the bytecode; co_filename is the filename from which the code was compiled; co_firstlineno is the first line number of the function; co_lnotab is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); co_stacksize is the required stack size (including local variables); co_flags is an integer encoding a number of flags for the interpreter.

The following flag bits are defined for co_flags: bit 0x04 is set if the function uses the *arguments syntax to accept an arbitrary number of positional arguments; bit 0x08 is set if the
function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator.

Future feature declarations (from `__future__ import division`) also use bits in `co_flags` to indicate whether a code object was compiled with a particular feature enabled: bit `0x2000` is set if the function was compiled with future division enabled; bits `0x10` and `0x1000` were used in earlier versions of Python.

Other bits in `co_flags` are reserved for internal use.

If a code object represents a function, the first item in `co_consts` is the documentation string of the function, or `None` if undefined.

Frame objects
Frame objects represent execution frames. They may occur in traceback objects (see below), and are also passed to registered trace functions.

Special read-only attributes: `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame; `f_code` is the code object being executed in this frame; `f_locals` is the dictionary used to look up local variables; `f_globals` is used for global variables; `f_builtins` is used for built-in (intrinsic) names; `f_lasti` gives the precise instruction (this is an index into the bytecode string of the code object).

Special writable attributes: `f_trace`, if not `None`, is a function
called for various events during code execution (this is used by the debugger). Normally an event is triggered for each new source line - this can be disabled by setting \texttt{f\_trace\_lines} to \texttt{False}.

Implementations \textit{may} allow per-opcode events to be requested by setting \texttt{f\_trace\_opcodes} to \texttt{True}. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.

\texttt{f\_lineno} is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a \texttt{Jump} command (aka \texttt{Set Next Statement}) by writing to \texttt{f\_lineno}.

Frame objects support one method:

\begin{verbatim}
frame.
\end{verbatim}

\texttt{clear()}

This method clears all references to local variables held by the frame. Also, if the frame belonged to a generator, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its traceback for later use).

\texttt{RuntimeError} is raised if the frame is currently executing.

\textit{New in version 3.4.}

Traceback objects

Traceback objects represent a stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling \texttt{types.TracebackType}.  

\textit{New in version 3.4.}
For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section `The try statement`.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the `tb_next` attributes should be linked to form a full stack trace.

Special read-only attributes: `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a `try` statement with no matching `except` clause or with a `finally` clause.

Special writable attribute: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

*Changed in version 3.7:* Traceback objects can now be explicitly instantiated from Python code, and the `tb_next`
attribute of existing instances can be updated.

Slice objects
Slice objects are used to represent slices for \texttt{\_\_getitem\_\_()} methods. They are also created by the built-in \texttt{slice()} function.

Special read-only attributes: start is the lower bound; stop is the upper bound; step is the step value; each is \texttt{None} if omitted. These attributes can have any type.

Slice objects support one method:

\texttt{slice.\_\_indices\_\_}(\texttt{self}, \texttt{length})
This method takes a single integer argument \texttt{length} and computes information about the slice that the slice object would describe if applied to a sequence of \texttt{length} items. It returns a tuple of three integers; respectively these are the \texttt{start} and \texttt{stop} indices and the \texttt{step} or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

Static method objects
Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in \texttt{staticmethod()} constructor.
Class method objects

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under “User-defined methods”. Class method objects are created by the built-in `classmethod()` constructor.
3.3. Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets `__iter__()` to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to `__getitem__()`).

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.3.1. Basic customization

```python
object.__new__(cls[, ...])
```

Called to create a new instance of class `cls`. `__new__()` is a static method (special-cased so you need not declare it as such) that
takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of \texttt{\_\_new\_()} should be the new object instance (usually an instance of \texttt{cls}).

Typical implementations create a new instance of the class by invoking the superclass's \texttt{\_\_new\_()} method using \texttt{super().\_\_new\_(cls[, ...,])} with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

If \texttt{\_\_new\_()} returns an instance of \texttt{cls}, then the new instance's \texttt{\_\_init\_()} method will be invoked like \texttt{\_\_init\_(self[, ...,])}, where \texttt{self} is the new instance and the remaining arguments are the same as were passed to \texttt{\_\_new\_()}.

If \texttt{\_\_new\_()} does not return an instance of \texttt{cls}, then the new instance's \texttt{\_\_init\_()} method will not be invoked.

\texttt{\_\_new\_()} is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

\texttt{object.\_\_init\_(self[, ...,])}

Called after the instance has been created (by \texttt{\_\_new\_()}), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an \texttt{\_\_init\_()} method, the derived class's \texttt{\_\_init\_()} method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example:
super().__init__([args...]).

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customize it), no non-None value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.

`object.__del__(self)`

Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

It is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. This is called object resurrection. It is implementation-dependent whether `__del__()` is called a second time when a resurrected object is about to be destroyed; the current CPython implementation only calls it once.

It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

**Note:** `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.

**CPython implementation detail:** It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by
the cyclic garbage collector. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

See also: Documentation for the gc module.

Warning: Due to the precarious circumstances under which __del__() methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to sys.stderr instead. In particular:
- __del__() can be invoked when arbitrary code is being executed, including from any arbitrary thread. If __del__() needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute __del__().
- __del__() can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to None. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the __del__() method is called.

object.__repr__(self)

Called by the repr() built-in function to compute the “official” string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If
this is not possible, a string of the form `<...some useful
description...>` should be returned. The return value must be
a string object. If a class defines `__repr__()` but not
`__str__()`, then `__repr__()` is also used when an “informal”
string representation of instances of that class is required.

This is typically used for debugging, so it is important that the
representation is information-rich and unambiguous.

```python
object.__str__(self)
```

Called by `str(object)` and the built-in functions `format()` and
`print()` to compute the “informal” or nicely printable string
representation of an object. The return value must be a string
object.

This method differs from `object.__repr__()` in that there is no
expectation that `__str__()` return a valid Python expression: a
more convenient or concise representation can be used.

The default implementation defined by the built-in type `object`
calls `object.__repr__()`.  

```python
object.__bytes__(self)
```

Called by `bytes` to compute a byte-string representation of an
object. This should return a bytes object.

```python
object.__format__(self, format_spec)
```

Called by the `format()` built-in function, and by extension,
evaluation of formatted string literals and the `str.format()`
method, to produce a “formatted” string representation of an object.
The `format_spec` argument is a string that contains a description
of the formatting options desired. The interpretation of the
format_spec argument is up to the type implementing __format__(), however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.


The return value must be a string object.

*Changed in version 3.4:* The __format__ method of object itself raises a TypeError if passed any non-empty string.

*Changed in version 3.7:* object.__format__(x, '') is now equivalent to str(x) rather than format(str(self), '').

object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)

These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows: x<y calls x.__lt__(y), x<=y calls x.__le__(y), x==y calls x.__eq__(y), x!=y calls x.__ne__(y), x>y calls x.__gt__(y), and x>=y calls x.__ge__(y).

A rich comparison method may return the singleton NotImplemented if it does not implement the operation for a given pair of arguments. By convention, False and True are returned for a successful comparison. However, these methods
can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an if statement), Python will call bool() on the value to determine if the result is true or false.

By default, __ne__() delegates to __eq__() and inverts the result unless it is NotImplemented. There are no other implied relationships among the comparison operators, for example, the truth of \((x<y \text{ or } x==y)\) does not imply \(x\leq y\). To automatically generate ordering operations from a single root operation, see functools.total_ordering().

See the paragraph on __hash__() for some important notes on creating hashable objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, __lt__() and __gt__() are each other’s reflection, __le__() and __ge__() are each other’s reflection, and __eq__() and __ne__() are their own reflection. If the operands are of different types, and right operand’s type is a direct or indirect subclass of the left operand’s type, the reflected method of the right operand has priority, otherwise the left operand’s method has priority. Virtual subclassing is not considered.

object.__hash__(self)

Called by built-in function hash() and for operations on members of hashed collections including set, frozenset, and dict. __hash__() should return an integer. The only required property is that objects which compare equal have the same hash value; it
is advised to mix together the hash values of the components of
the object that also play a part in comparison of objects by packing
them into a tuple and hashing the tuple. Example:

```python
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

**Note:** `hash()` truncates the value returned from an object's
custom `__hash__()` method to the size of a `Py_ssize_t`. This
is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit builds. If
an object's `__hash__()` must interoperate on builds of different
bit sizes, be sure to check the width on all supported builds. An
easy way to do this is with `python -c "import sys; print(sys.hash_info.width)"`.

If a class does not define an `__eq__()` method it should not
define a `__hash__()` operation either; if it defines `__eq__()` but
not `__hash__()`, its instances will not be usable as items in
hashable collections. If a class defines mutable objects and
implements an `__eq__()` method, it should not implement
`__hash__()`, since the implementation of hashable collections
requires that a key's hash value is immutable (if the object's hash
value changes, it will be in the wrong hash bucket).

User-defined classes have `__eq__()` and `__hash__()` methods
by default; with them, all objects compare unequal (except with
themselves) and `x.__hash__()` returns an appropriate value
such that `x == y` implies both that `x is y` and `hash(x) ==
hash(y)`.

A class that overrides `__eq__()` and does not define
__hash__() will have its __hash__() implicitly set to None. When the __hash__() method of a class is None, instances of the class will raise an appropriate TypeError when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking isinstance(obj, collections.abc.Hashable).

If a class that overrides __eq__() needs to retain the implementation of __hash__() from a parent class, the interpreter must be told this explicitly by setting __hash__ = <ParentClass>.__hash__.

If a class that does not override __eq__() wishes to suppress hash support, it should include __hash__ = None in the class definition. A class which defines its own __hash__() that explicitly raises a TypeError would be incorrectly identified as hashable by an isinstance(obj, collections.abc.Hashable) call.

**Note:** By default, the __hash__() values of str, bytes and datetime objects are “salted” with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

This is intended to provide protection against a denial-of-service caused by carefully-chosen inputs that exploit the worst case performance of a dict insertion, O(n^2) complexity. See http://www.ocert.org/advisories/ocert-2011-003.html for details.

Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).
See also PYTHONHASHSEED.

*Changed in version 3.3:* Hash randomization is enabled by default.

**object.** __bool__(**self**)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`. When this method is not defined, **len**(()) is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither **len**(()) nor **bool**(()), all its instances are considered true.

### 3.3.2. Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of `x.name`) for class instances.

**object.** __getattr__(**self**, **name**)

Called when the default attribute access fails with an `AttributeError` (either __getattribute__() raises an `AttributeError` because `name` is not an instance attribute or an attribute in the class tree for **self**; or __get__() of a `name` property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception.

Note that if the attribute is found through the normal mechanism, __getattr__() is not called. (This is an intentional asymmetry between __getattr__() and __setattr__().) This is done both for efficiency reasons and because otherwise
__getattr__() would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the __getattribute__() method below for a way to actually get total control over attribute access.

object.__getattribute__(self, name)
Called unconditionally to implement attribute accesses for instances of the class. If the class also defines __getattr__(), the latter will not be called unless __getattribute__() either calls it explicitly or raises an AttributeError. This method should return the (computed) attribute value or raise an AttributeError exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, object.__getattribute__(self, name).

Note:  This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See Special method lookup.

object.__setattr__(self, name, value)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). name is the attribute name, value is the value to be assigned to it.

If __setattr__() wants to assign to an instance attribute, it should call the base class method with the same name, for example, object.__setattr__(self, name, value).
object.__delattr__(self, name)
Like __setattr__() but for attribute deletion instead of assignment. This should only be implemented if del obj.name is meaningful for the object.

object.__dir__(self)
Called when dir() is called on the object. A sequence must be returned. dir() converts the returned sequence to a list and sorts it.

3.3.2.1. Customizing module attribute access

Special names __getattr__ and __dir__ can be also used to customize access to module attributes. The __getattr__ function at the module level should accept one argument which is the name of an attribute and return the computed value or raise an AttributeError. If an attribute is not found on a module object through the normal lookup, i.e. object.__getattribute__(), then __getattr__ is searched in the module __dict__ before raising an AttributeError. If found, it is called with the attribute name and the result is returned.

The __dir__ function should accept no arguments, and return a list of strings that represents the names accessible on module. If present, this function overrides the standard dir() search on a module.

For a more fine grained customization of the module behavior (setting attributes, properties, etc.), one can set the __class__ attribute of a module object to a subclass of types.ModuleType. For example:

```python
import sys
from types import ModuleType
```
class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule

Note: Defining module __getattr__ and setting module __class__ only affect lookups made using the attribute access syntax – directly accessing the module globals (whether by code within the module, or via a reference to the module’s globals dictionary) is unaffected.

Changed in version 3.5: __class__ module attribute is now writable.

New in version 3.7: __getattr__ and __dir__ module attributes.

See also:

PEP 562 - Module __getattr__ and __dir__
    Describes the __getattr__ and __dir__ functions on modules.

3.3.2.2. Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called descriptor class) appears in an owner class (the descriptor must be in either the owner’s class dictionary or in the class dictionary for one of its parents). In the
examples below, “the attribute” refers to the attribute whose name is the key of the property in the owner class’ \_\_dict\_.

object.\_\_get\_(self, instance, owner)
Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). \_owner\_ is always the owner class, while \_instance\_ is the instance that the attribute was accessed through, or \_None\_ when the attribute is accessed through the \_owner\_. This method should return the (computed) attribute value or raise an \_\_AttributeError\_ exception.

object.\_\_set\_(self, instance, value)
Called to set the attribute on an instance \_instance\_ of the owner class to a new value, \_value\_.

object.\_\_delete\_(self, instance)
Called to delete the attribute on an instance \_instance\_ of the owner class.

object.\_\_set_name\_(self, owner, name)
Called at the time the owning class \_owner\_ is created. The descriptor has been assigned to \_name\_.

\_New in version 3.6.\_

The attribute \_\_objclass\_ is interpreted by the \_\_inspect\_ module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).
3.3.2.3. Invoking Descriptors

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol: \texttt{\_\_get\_\_()}, \texttt{\_\_set\_\_()}, and \texttt{\_\_delete\_\_()}. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, \texttt{a.x} has a lookup chain starting with \texttt{a.\_\_dict\_\_['x']}, then \texttt{type(a).\_\_dict\_\_['x']}, and continuing through the base classes of \texttt{type(a)} excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, \texttt{a.x}. How the arguments are assembled depends on \texttt{a}:

Direct Call

The simplest and least common call is when user code directly invokes a descriptor method: \texttt{\_\_get\_\_(a)}.

Instance Binding

If binding to an object instance, \texttt{a.x} is transformed into the call: \texttt{type(a).\_\_dict\_\_['x'].\_\_get\_\_(a, type(a))}.

Class Binding

If binding to a class, \texttt{A.x} is transformed into the call: \texttt{A.\_\_dict\_\_['x'].\_\_get\_\_(None, A)}. 
Super Binding

If \(a\) is an instance of \texttt{super}, then the binding \texttt{super(B, obj).m()} searches \texttt{obj.__class__.__mro__} for the base class \texttt{A} immediately preceding \texttt{B} and then invokes the descriptor with the call: \(A._\_dict_\_['m']._\_get__(obj, obj._\_class__).\)

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. A descriptor can define any combination of \texttt{__get__()}, \texttt{__set__()} and \texttt{__delete__()}. If it does not define \texttt{__get__()}, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines \texttt{__set__()} and/or \texttt{__delete__()}, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both \texttt{__get__()} and \texttt{__set__()}, while non-data descriptors have just the \texttt{__get__()} method. Data descriptors with \texttt{__set__()} and \texttt{__get__()} defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including \texttt{staticmethod()} and \texttt{classmethod()}) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The \texttt{property()} function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

3.3.2.4. \_slots\_

\_slots\_ allow us to explicitly declare data members (like properties)
and deny the creation of __dict__ and __weakref__ (unless explicitly declared in __slots__ or available in a parent.)

The space saved over using __dict__ can be significant. Attribute lookup speed can be significantly improved as well.

object.__slots__

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. __slots__ reserves space for the declared variables and prevents the automatic creation of __dict__ and __weakref__ for each instance.

3.3.2.4.1. Notes on using __slots__

- When inheriting from a class without __slots__, the __dict__ and __weakref__ attribute of the instances will always be accessible.
- Without a __dict__ variable, instances cannot be assigned new variables not listed in the __slots__ definition. Attempts to assign to an unlisted variable name raises AttributeError. If dynamic assignment of new variables is desired, then add '__dict__' to the sequence of strings in the __slots__ declaration.
- Without a __weakref__ variable for each instance, classes defining __slots__ do not support weak references to its instances. If weak reference support is needed, then add '__weakref__' to the sequence of strings in the __slots__ declaration.
- __slots__ are implemented at the class level by creating descriptors (Implementing Descriptors) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by __slots__; otherwise, the class attribute would overwrite the descriptor assignment.
- The action of a __slots__ declaration is not limited to the class
where it is defined. __slots__ declared in parents are available in child classes. However, child subclasses will get a __dict__ and __weakref__ unless they also define __slots__ (which should only contain names of any additional slots).

- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- Nonempty __slots__ does not work for classes derived from “variable-length” built-in types such as int, bytes and tuple.
- Any non-string iterable may be assigned to __slots__. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.
- __class__ assignment works only if both classes have the same __slots__.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise TypeError.

### 3.3.3. Customizing class creation

Whenever a class inherits from another class, __init_subclass__ is called on that class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they're applied to, __init_subclass__ solely applies to future subclasses of the class defining the method.

```python
classmethod object.__init_subclass__(cls)
```

This method is called whenever the containing class is subclassed. cls is then the new subclass. If defined as a normal instance
method, this method is implicitly converted to a class method.

Keyword arguments which are given to a new class are passed to
the parent's class `__init_subclass__`. For compatibility with
other classes using `__init_subclass__`, one should take out
the needed keyword arguments and pass the others over to the
base class, as in:

```python
class Philosopher:
    def __init_subclass__(cls, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name):
    pass
```

The default implementation `object.__init_subclass__` does
nothing, but raises an error if it is called with any arguments.

**Note:** The metaclass hint `metaclass` is consumed by the rest
of the type machinery, and is never passed to
`__init_subclass__` implementations. The actual metaclass
(rather than the explicit hint) can be accessed as `type(cls)`.

_New in version 3.6._

### 3.3.3.1. Metaclasses

By default, classes are constructed using `type()`. The class body is
executed in a new namespace and the class name is bound locally to
the result of `type(name, bases, namespace)`.

The class creation process can be customized by passing the
metaclass keyword argument in the class definition line, or by inheriting from an existing class that included such an argument. In the following example, both MyClass and MySubclass are instances of Meta:

```python
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Any other keyword arguments that are specified in the class definition are passed through to all metaclass operations described below.

When a class definition is executed, the following steps occur:

- MRO entries are resolved
- the appropriate metaclass is determined
- the class namespace is prepared
- the class body is executed
- the class object is created

3.3.3.2. Resolving MRO entries

If a base that appears in class definition is not an instance of type, then an __mro_entries__ method is searched on it. If found, it is called with the original bases tuple. This method must return a tuple of classes that will be used instead of this base. The tuple may be empty, in such case the original base is ignored.

See also: PEP 560 - Core support for typing module and generic
3.3.3.3. Determining the appropriate metaclass

The appropriate metaclass for a class definition is determined as follows:

- if no bases and no explicit metaclass are given, then `type()` is used
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass
- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of *all* of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

3.3.3.4. Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition).

If the metaclass has no `__prepare__` attribute, then the class namespace is initialised as an empty ordered mapping.

See also:
3.3.3.5. Executing the class body

The class body is executed (approximately) as `exec(body, globals(), namespace)`. The key difference from a normal call to `exec()` is that lexical scoping allows the class body (including any methods) to reference names from the current and outer scopes when the class definition occurs inside a function.

However, even when the class definition occurs inside the function, methods defined inside the class still cannot see names defined at the class scope. Class variables must be accessed through the first parameter of instance or class methods, or through the implicit lexically scoped `__class__` reference described in the next section.

3.3.3.6. Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling `metaclass(name, bases, namespace, **kwds)` (the additional keywords passed here are the same as those passed to `__prepare__`).

This class object is the one that will be referenced by the zero-argument form of `super()`. `__class__` is an implicit closure reference created by the compiler if any methods in a class body refer to either `__class__` or `super`. This allows the zero argument form of `super()` to correctly identify the class being defined based on lexical scoping, while the class or instance that was used to make the current call is identified based on the first argument passed to the method.
**CPython implementation detail:** In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `DeprecationWarning` in Python 3.6, and a `RuntimeError` in Python 3.8.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customisation steps are invoked after creating the class object:

- first, `type.__new__` collects all of the descriptors in the class namespace that define a `__set_name__()` method;
- second, all of these `__set_name__` methods are called with the class being defined and the assigned name of that particular descriptor; and
- finally, the `__init_subclass__()` hook is called on the immediate parent of the new class in its method resolution order.

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the `__dict__` attribute of the class object.

**See also:**

**PEP 3135 - New super**

Describes the implicit `__class__` closure reference
3.3.3.7. Uses for metaclasses

The potential uses for metaclasses are boundless. Some ideas that have been explored include enum, logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.3.4. Customizing instance and subclass checks

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as “virtual base classes” to any class or type (including built-in types), including other ABCs.

```python
class.__instancecheck__(self, instance)
```

Return true if `instance` should be considered a (direct or indirect) instance of `class`. If defined, called to implement `isinstance(instance, class)`.

```python
class.__subclasscheck__(self, subclass)
```

Return true if `subclass` should be considered a (direct or indirect) subclass of `class`. If defined, called to implement `issubclass(subclass, class)`.

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.
3.3.5. Emulating generic types

One can implement the generic class syntax as specified by PEP 484 (for example List[int]) by defining a special method

```python
classmethod object.__class_getitem__(cls, key)
```

Return an object representing the specialization of a generic class by type arguments found in `key`.

This method is looked up on the class object itself, and when defined in the class body, this method is implicitly a class method. Note, this mechanism is primarily reserved for use with static type hints, other usage is discouraged.

See also: PEP 560 - Core support for typing module and generic types

3.3.6. Emulating callable objects

```python
object.__call__(self[, args...])
```

Called when the instance is “called” as a function; if this method is defined, `x(arg1, arg2, ...)` is a shorthand for
x.__call__(arg1, arg2, ...).

3.3.7. Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers \( k \) for which \( 0 \leq k < N \) where \( N \) is the length of the sequence, or slice objects, which define a range of items. It is also recommended that mappings provide the methods keys(), values(), items(), get(), clear(), setdefault(), pop(), popitem(), copy(), and update() behaving similar to those for Python's standard dictionary objects. The collections.abc module provides a MutableMapping abstract base class to help create those methods from a base set of __getitem__(), __setitem__(), __delitem__(), and keys().

Mutable sequences should provide methods append(), count(), index(), extend(), insert(), pop(), remove(), reverse() and sort(), like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods __add__(), __radd__(), __iadd__(), __mul__(), __rmul__(), and __imul__() described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the __contains__() method to allow efficient use of the in operator; for mappings, in should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the
\_\_iter\_\_() method to allow efficient iteration through the container; for mappings, \_\_iter\_\_() should be the same as keys(); for sequences, it should iterate through the values.

**object.\_\_len\_\_(self)**

Called to implement the built-in function `len()`. Should return the length of the object, an integer \( \geq 0 \). Also, an object that doesn't define a \_\_bool\_\_( ) method and whose \_\_len\_\_( ) method returns zero is considered to be false in a Boolean context.

**CPython implementation detail:** In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a \_\_bool\_\_( ) method.

**object.\_\_length\_hint\_\_(self)**

Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer \( \geq 0 \). This method is purely an optimization and is never required for correctness.

*New in version 3.4.*

**Note:** Slicing is done exclusively with the following three methods. A call like

```python
a[1:2] = b
```

is translated to
a[slice(1, 2, None)] = b

and so forth. Missing slice items are always filled in with None.

object.__getitem__(self, key)
Called to implement evaluation of self[key]. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the __getitem__() method. If key is of an inappropriate type, TypeError may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), IndexError should be raised. For mapping types, if key is missing (not in the container), KeyError should be raised.

Note: for loops expect that an IndexError will be raised for illegal indexes to allow proper detection of the end of the sequence.

object.__setitem__(self, key, value)
Called to implement assignment to self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper key values as for the __getitem__() method.

object.__delitem__(self, key)
Called to implement deletion of self[key]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements
can be removed from the sequence. The same exceptions should be raised for improper key values as for the \texttt{\_\_getitem\_\_()} method.

\texttt{object.\_\_missing\_\_(self, key)}

Called by \texttt{dict.\_\_getitem\_\_()} to implement \texttt{self[key]} for dict subclasses when key is not in the dictionary.

\texttt{object.\_\_iter\_\_(self)}

This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see \texttt{Iterator Types}.

\texttt{object.\_\_reversed\_\_(self)}

Called (if present) by the \texttt{reversed()} built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the \texttt{\_\_reversed\_\_()} method is not provided, the \texttt{reversed()} built-in will fall back to using the sequence protocol (\texttt{\_\_len\_\_()} and \texttt{\_\_getitem\_\_()}). Objects that support the sequence protocol should only provide \texttt{\_\_reversed\_\_()} if they can provide an implementation that is more efficient than the one provided by \texttt{reversed()}.

The membership test operators (\texttt{in} and \texttt{not in}) are normally implemented as an iteration through a sequence. However, container
objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

```python
object.__contains__(self, item)
```

Called to implement membership test operators. Should return true if `item` is in `self`, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don’t define `__contains__()`, the membership test first tries iteration via `__iter__()`, then the old sequence iteration protocol via `__getitem__()`, see this section in the language reference.

### 3.3.8. Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

```python
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
```
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, & , ^, |). For instance, to evaluate the expression \( x + y \), where \( x \) is an instance of a class that has an \_\_add\_\_( ) method, \( x.__add__(y) \) is called. The \_\_divmod\_\_( ) method should be the equivalent to using \_\_floordiv\_\_( ) and \_\_mod\_\_( ); it should not be related to \_\_truediv\_\_( ). Note that \_\_pow\_\_( ) should be defined to accept an optional third argument if the ternary version of the built-in \texttt{pow()} function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return \texttt{NotImplemented}.

object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other)
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)

These methods are called to implement the binary arithmetic
operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation [3] and the operands are of different types. [4] For instance, to evaluate the expression \( x - y \), where \( y \) is an instance of a class that has an \(__rsub__()\) method, \( y.__rsub__(x) \) is called if \( x.__sub__(y) \) returns \textit{NotImplemented}.

Note that ternary \texttt{pow()} will not try calling \texttt{__rpow__()} (the coercion rules would become too complicated).

**Note:** If the right operand's type is a subclass of the left operand's type and that subclass provides the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

\[
\begin{align*}
\text{object.}\_\text{imod}\_&(\text{self, other}) \\
\text{object.}\_\text{imul}\_&(\text{self, other}) \\
\text{object.}\_\text{imatmul}\_&(\text{self, other}) \\
\text{object.}\_\text{itruediv}\_&(\text{self, other}) \\
\text{object.}\_\text{ifloordiv}\_&(\text{self, other}) \\
\text{object.}\_\text{ipow}\_&(\text{self, other}[, \text{modulo}]) \\
\text{object.}\_\text{ilshift}\_&(\text{self, other}) \\
\text{object.}\_\text{irshift}\_&(\text{self, other}) \\
\text{object.}\_\text{iand}\_&(\text{self, other}) \\
\text{object.}\_\text{ixor}\_&(\text{self, other}) \\
\text{object.}\_\text{ior}\_&(\text{self, other})
\end{align*}
\]

These methods are called to implement the augmented arithmetic
assignments (+=, -=, *=, @=, /=, /=, %=, **=, <<=, >>=, &=, ^=, |=). These methods should attempt to do the operation in-place (modifying self) and return the result (which could be, but does not have to be, self). If a specific method is not defined, the augmented assignment falls back to the normal methods. For instance, if x is an instance of a class with an __iadd__() method, x += y is equivalent to x = x.__iadd__(y). Otherwise, x.__add__(y) and y.__radd__(x) are considered, as with the evaluation of x + y. In certain situations, augmented assignment can result in unexpected errors (see Why does a_tuple[i] += [item] raise an exception when the addition works?), but this behavior is in fact part of the data model.

object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)

Called to implement the unary arithmetic operations (-, +, abs() and ~).

object.__complex__(self)
object.__int__(self)
object.__float__(self)

Called to implement the built-in functions complex(), int() and float(). Should return a value of the appropriate type.

object.__index__(self)

Called to implement operator.index(), and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in bin(), hex() and oct() functions). Presence of this method indicates that the numeric
object is an integer type. Must return an integer.

**Note:** In order to have a coherent integer type class, when \_\_index\_(\) is defined \_\_int\_(\) should also be defined, and both should return the same value.

\_\_round\_(self[, ndigits])
\_\_trunc\_(self)
\_\_floor\_(self)
\_\_ceil\_(self)

Called to implement the built-in function `round()` and `math` functions `trunc()`, `floor()` and `ceil()`. Unless `ndigits` is passed to \_\_round\_(\) all these methods should return the value of the object truncated to an `Integral` (typically an `int`).

If \_\_int\_(\) is not defined then the built-in function `int()` falls back to \_\_trunc\_(\).

3.3.9. With Statement Context Managers

A *context manager* is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.
For more information on context managers, see Context Manager Types.

object.__enter__(self)

Enter the runtime context related to this object. The with statement will bind this method’s return value to the target(s) specified in the as clause of the statement, if any.

object.__exit__(self, exc_type, exc_value, traceback)

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be None.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that __exit__() methods should not reraise the passed-in exception; this is the caller’s responsibility.

See also:

PEP 343 - The “with” statement

The specification, background, and examples for the Python with statement.

3.3.10. Special method lookup

For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object’s type, not in the
object's instance dictionary. That behaviour is the reason why the following code raises an exception:

```python
>>> class C:
    ...
    pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as `__hash__()` and `__repr__()` that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```python
>>> 1.__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as ‘metaclass confusion’, and is avoided by bypassing the instance when looking up special methods:

```python
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```
In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the `__getattribute__()` method even of the object's metaclass:

```python
>>> class Meta(type):
...    def __getattribute__(*args):
...        print("Metaclass getattribute invoked")
...        return type.__getattribute__(*args)
...    
>>> class C(object, metaclass=Meta):
...    def __len__(self):
...        return 10
...    def __getattribute__(*args):
...        print("Class getattribute invoked")
...        return object.__getattribute__(*args)
...    
>>> c = C()
>>> c.__len__()  # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)  # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)  # Implicit lookup
10
```

Bypassing the `__getattribute__()` machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method `must` be set on the class object itself in order to be consistently invoked by the interpreter).
3.4. Coroutines

3.4.1. Awaitable Objects

An *awaitable* object generally implements an `__await__()` method. *Coroutine* objects returned from `async def` functions are awaitable.

**Note:** The *generator iterator* objects returned from generators decorated with `types.coroutine()` or `asyncio.coroutine()` are also awaitable, but they do not implement `__await__()`. 

```python
object.__await__(self)
```

Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, `asyncio.Future` implements this method to be compatible with the `await` expression.

*New in version 3.5.*

**See also:** PEP 492 for additional information about awaitable objects.

3.4.2. Coroutine Objects

*Coroutine* objects are *awaitable* objects. A coroutine’s execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.
Coroutines also have the methods listed below, which are analogous to those of generators (see Generator-iterator methods). However, unlike generators, coroutines do not directly support iteration.

*Changed in version 3.5.2:* It is a `RuntimeError` to await on a coroutine more than once.

coroutine.**send**(value)

Starts or resumes execution of the coroutine. If `value` is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If `value` is not `None`, this method delegates to the `send()` method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__() return value, described above.

coroutine.**throw**(type[, value[, traceback]])

Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__() return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

coroutine.**close**()

Causes the coroutine to clean itself up and exit. If the coroutine is suspended, this method first delegates to the `close()` method of the iterator that caused the coroutine to suspend, if it has such a method. Then it raises `GeneratorExit` at the suspension point, causing the coroutine to immediately clean itself up. Finally, the
Coroutine is marked as having finished executing, even if it was never started.

Coroutine objects are automatically closed using the above process when they are about to be destroyed.

### 3.4.3. Asynchronous Iterators

An *asynchronous iterator* can call asynchronous code in its `__anext__` method.

Asynchronous iterators can be used in an `async for` statement.

```
object.__aiter__(self)
    Must return an *asynchronous iterator* object.

object.__anext__(self)
    Must return an *awaitable* resulting in a next value of the iterator. Should raise a `StopAsyncIteration` error when the iteration is over.
```

An example of an asynchronous iterable object:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
```
New in version 3.5.

Changed in version 3.7: Prior to Python 3.7, `__aiter__` could return an `awaitable` that would resolve to an asynchronous iterator.

Starting with Python 3.7, `__aiter__` must return an asynchronous iterator object. Returning anything else will result in a `TypeError` error.

### 3.4.4. Asynchronous Context Managers

An asynchronous context manager is a context manager that is able to suspend execution in its `__aenter__` and `__aexit__` methods.

Asynchronous context managers can be used in an `async with` statement.

```python
object.__aenter__(self)

This method is semantically similar to the `__enter__()`, with only difference that it must return an `awaitable`.

object.__aexit__(self, exc_type, exc_value, traceback)

This method is semantically similar to the `__exit__()`, with only difference that it must return an `awaitable`.

An example of an asynchronous context manager class:

```python
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')
```
async def __aexit__(self, exc_type, exc, tb):
    await log('exiting context')

New in version 3.5.

Footnotes

It is possible in some cases to change an object’s type, under certain controlled conditions. It generally isn’t a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

The __hash__(), __iter__(), __reversed__(), and __contains__() methods have special handling for this; others will still raise a TypeError, but may do so by relying on the behavior that None is not callable.

“Does not support” here means that the class has no such method, or the method returns NotImplemented. Do not set the method to None if you want to force fallback to the right operand’s reflected method—that will instead have the opposite effect of explicitly blocking such fallback.

For operands of the same type, it is assumed that if the non-reflected method (such as __add__()) fails the operation is not supported, which is why the reflected method is not called.
4. Execution model
4.1. Structure of a program

A Python program is constructed from code blocks. A block is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the `{-c` option) is a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

A code block is executed in an execution frame. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block’s execution has completed.
4.2. Naming and binding

4.2.1. Binding of names

*Names* refer to objects. Names are introduced by name binding operations.

The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, `for` loop header, or after as in a `with` statement or `except` clause. The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name is bound in a block, it is a local variable of that block, unless declared as `nonlocal` or `global`. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

Each occurrence of a name in the program text refers to the *binding* of that name established by the following name resolution rules.
4.2.2. Resolution of names

A scope defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's environment.

When a name is not found at all, a NameError exception is raised. If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an UnboundLocalError exception is raised. UnboundLocalError is a subclass of NameError.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module builtins. The global namespace is
searched first. If the name is not found there, the builtins namespace is searched. The **global** statement must precede all uses of the name.

The **global** statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

The **nonlocal** statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. **SyntaxError** is raised at compile time if the given name does not exist in any enclosing function scope.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called **__main__**.

Class definition blocks and arguments to **exec()** and **eval()** are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:

```python
class A:
    a = 42
    b = list(a + i for i in range(10))
```
4.2.3. Builtins and restricted execution

**CPython implementation detail:** Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should import the `builtins` module and modify its attributes appropriately.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module’s dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `builtins`; when in any other module, `__builtins__` is an alias for the dictionary of the `builtins` module itself.

4.2.4. Interaction with dynamic features

Name resolution of free variables occurs at runtime, not at compile time. This means that the following code will print 42:

```python
i = 10
def f():
    print(i)
i = 42
f()
```

The `eval()` and `exec()` functions do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace. [1] The `exec()` and `eval()` functions have optional arguments to override the global and local namespace. If only one namespace is
specified, it is used for both.
4.3. Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the raise statement. Exception handlers are specified with the try ... except statement. The finally clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the “termination” model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is SystemExit.

Exceptions are identified by class instances. The except clause is selected depending on the class of the instance: it must reference the class of the instance or a base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.
Note: Exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the try statement in section The try statement and raise statement in section The raise statement.

Footnotes

This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

[1]
5. The import system

Python code in one module gains access to the code in another module by the process of importing it. The import statement is the most common way of invoking the import machinery, but it is not the only way. Functions such as importlib.import_module() and built-in __import__() can also be used to invoke the import machinery.

The import statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope. The search operation of the import statement is defined as a call to the __import__() function, with the appropriate arguments. The return value of __import__() is used to perform the name binding operation of the import statement. See the import statement for the exact details of that name binding operation.

A direct call to __import__() performs only the module search and, if found, the module creation operation. While certain side-effects may occur, such as the importing of parent packages, and the updating of various caches (including sys.modules), only the import statement performs a name binding operation.

When an import statement is executed, the standard builtin __import__() function is called. Other mechanisms for invoking the import system (such as importlib.import_module()) may choose to bypass __import__() and use their own solutions to implement import semantics.

When a module is first imported, Python searches for the module and if found, it creates a module object [1], initializing it. If the named
module cannot be found, a `ModuleNotFoundError` is raised. Python implements various strategies to search for the named module when the import machinery is invoked. These strategies can be modified and extended by using various hooks described in the sections below.

*Changed in version 3.3:* The import system has been updated to fully implement the second phase of [PEP 302](https://www.python.org/dev/peps/pep-302/). There is no longer any implicit import machinery - the full import system is exposed through `sys.meta_path`. In addition, native namespace package support has been implemented (see [PEP 420](https://www.python.org/dev/peps/pep-420/)).
5.1. *importlib*

The *importlib* module provides a rich API for interacting with the import system. For example, `importlib.import_module()` provides a recommended, simpler API than built-in `__import__()` for invoking the import machinery. Refer to the *importlib* library documentation for additional detail.
5.2. Packages

Python has only one type of module object, and all modules are of this type, regardless of whether the module is implemented in Python, C, or something else. To help organize modules and provide a naming hierarchy, Python has a concept of packages.

You can think of packages as the directories on a file system and modules as files within directories, but don’t take this analogy too literally since packages and modules need not originate from the file system. For the purposes of this documentation, we’ll use this convenient analogy of directories and files. Like file system directories, packages are organized hierarchically, and packages may themselves contain subpackages, as well as regular modules.

It’s important to keep in mind that all packages are modules, but not all modules are packages. Or put another way, packages are just a special kind of module. Specifically, any module that contains a `__path__` attribute is considered a package.

All modules have a name. Subpackage names are separated from their parent package name by dots, akin to Python’s standard attribute access syntax. Thus you might have a module called `sys` and a package called `email`, which in turn has a subpackage called `email.mime` and a module within that subpackage called `email.mime.text`.

5.2.1. Regular packages

Python defines two types of packages, regular packages and namespace packages. Regular packages are traditional packages as
they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an \__init__.py file. When a regular package is imported, this \__init__.py file is implicitly executed, and the objects it defines are bound to names in the package's namespace. The \__init__.py file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

For example, the following file system layout defines a top level parent package with three subpackages:

```
parent/
    \__init__.py
one/
    \__init__.py
two/
    \__init__.py
three/
    \__init__.py
```

Importing parent.one will implicitly execute parent/\__init__.py and parent/one/\__init__.py. Subsequent imports of parent.two or parent.three will execute parent/two/\__init__.py and parent/three/\__init__.py respectively.

### 5.2.2. Namespace packages

A namespace package is a composite of various portions, where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python
searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.

Namespace packages do not use an ordinary list for their `__path__` attribute. They instead use a custom iterable type which will automatically perform a new search for package portions on the next import attempt within that package if the path of their parent package (or `sys.path` for a top level package) changes.

With namespace packages, there is no `parent/__init__.py` file. In fact, there may be multiple `parent` directories found during import search, where each one is provided by a different portion. Thus `parent/one` may not be physically located next to `parent/two`. In this case, Python will create a namespace package for the top-level `parent` package whenever it or one of its subpackages is imported.

See also PEP 420 for the namespace package specification.
5.3. Searching

To begin the search, Python needs the **fully qualified** name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the `import` statement, or from the parameters to the `importlib.import_module()` or `__import__()` functions.

This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a `ModuleNotFoundError` is raised.

5.3.1. The module cache

The first place checked during import search is `sys.modules`. This mapping serves as a cache of all modules that have been previously imported, including the intermediate paths. So if `foo.bar.baz` was previously imported, `sys.modules` will contain entries for `foo`, `foo.bar`, and `foo.bar.baz`. Each key will have as its value the corresponding module object.

During import, the module name is looked up in `sys.modules` and if present, the associated value is the module satisfying the import, and the process completes. However, if the value is `None`, then a `ModuleNotFoundError` is raised. If the module name is missing, Python will continue searching for the module.

`sys.modules` is writable. Deleting a key may not destroy the
associated module (as other modules may hold references to it), but it will invalidate the cache entry for the named module, causing Python to search anew for the named module upon its next import. The key can also be assigned to `None`, forcing the next import of the module to result in a `ModuleNotFoundError`.

Beware though, as if you keep a reference to the module object, invalidate its cache entry in `sys.modules`, and then re-import the named module, the two module objects will *not* be the same. By contrast, `importlib.reload()` will reuse the *same* module object, and simply reinitialise the module contents by rerunning the module's code.

### 5.3.2. Finders and loaders

If the named module is not found in `sys.modules`, then Python’s import protocol is invoked to find and load the module. This protocol consists of two conceptual objects, finders and loaders. A finder’s job is to determine whether it can find the named module using whatever strategy it knows about. Objects that implement both of these interfaces are referred to as importers - they return themselves when they find that they can load the requested module.

Python includes a number of default finders and importers. The first one knows how to locate built-in modules, and the second knows how to locate frozen modules. A third default finder searches an import path for modules. The import path is a list of locations that may name file system paths or zip files. It can also be extended to search for any locatable resource, such as those identified by URLs.

The import machinery is extensible, so new finders can be added to extend the range and scope of module searching.
Finders do not actually load modules. If they can find the named module, they return a *module spec*, an encapsulation of the module's import-related information, which the import machinery then uses when loading the module.

The following sections describe the protocol for finders and loaders in more detail, including how you can create and register new ones to extend the import machinery.

*Changed in version 3.4:* In previous versions of Python, finders returned *loaders* directly, whereas now they return module specs which *contain* loaders. Loaders are still used during import but have fewer responsibilities.

### 5.3.3. Import hooks

The import machinery is designed to be extensible; the primary mechanism for this are the *import hooks*. There are two types of import hooks: *meta hooks* and *import path hooks*.

Meta hooks are called at the start of import processing, before any other import processing has occurred, other than `sys.modules` cache look up. This allows meta hooks to override `sys.path` processing, frozen modules, or even built-in modules. Meta hooks are registered by adding new finder objects to `sys.meta_path`, as described below.

Import path hooks are called as part of `sys.path` (or `package.__path__`) processing, at the point where their associated path item is encountered. Import path hooks are registered by adding new callables to `sys.path_hooks` as described below.
5.3.4. The meta path

When the named module is not found in `sys.modules`, Python next searches `sys.meta_path`, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called `find_spec()` which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

If the meta path finder knows how to handle the named module, it returns a spec object. If it cannot handle the named module, it returns `None`. If `sys.meta_path` processing reaches the end of its list without returning a spec, then a `ModuleNotFoundError` is raised. Any other exceptions raised are simply propagated up, aborting the import process.

The `find_spec()` method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example `foo.bar.baz`. The second argument is the path entries to use for the module search. For top-level modules, the second argument is `None`, but for submodules or subpackages, the second argument is the value of the parent package's `__path__` attribute. If the appropriate `__path__` attribute cannot be accessed, a `ModuleNotFoundError` is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

The meta path may be traversed multiple times for a single import request. For example, assuming none of the modules involved has
already been cached, importing foo.bar.baz will first perform a top level import, calling `mpf.find_spec("foo", None, None)` on each meta path finder (mpf). After foo has been imported, foo.bar will be imported by traversing the meta path a second time, calling `mpf.find_spec("foo.bar", foo.__path__, None)`. Once foo.bar has been imported, the final traversal will call `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Some meta path finders only support top level imports. These importers will always return None when anything other than None is passed as the second argument.

Python’s default `sys.meta_path` has three meta path finders, one that knows how to import built-in modules, one that knows how to import frozen modules, and one that knows how to import modules from an import path (i.e. the path based finder).

*Changed in version 3.4:* The `find_spec()` method of meta path finders replaced `find_module()`, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement `find_spec()`.
5.4. Loading

If and when a module spec is found, the import machinery will use it (and the loader it contains) when loading the module. Here is an approximation of what happens during the loading portion of import:

```python
module = None
if spec.loader is not None and hasattr(spec.loader, 'exec_module',
   '# It is assumed 'exec_module' will also be defined
   module = spec.loader.create_module(spec)
if module is None:
   module = ModuleType(spec.name)
   # The import-related module attributes get set here:
   _init_module_attrs(spec, module)

if spec.loader is None:
   if spec.submodule_search_locations is not None:
      # namespace package
      sys.modules[spec.name] = module
   else:
      # unsupported
      raise ImportError
elif not hasattr(spec.loader, 'exec_module'):
   module = spec.loader.load_module(spec.name)
   # Set __loader__ and __package__ if missing.
else:
   sys.modules[spec.name] = module
try:
   spec.loader.exec_module(module)
except BaseException:
   try:
      del sys.modules[spec.name]
   except KeyError:
      pass
raise
```
Note the following details:

- If there is an existing module object with the given name in `sys.modules`, import will have already returned it.
- The module will exist in `sys.modules` before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to `sys.modules` beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module – and only the failing module – gets removed from `sys.modules`. Any module already in the `sys.modules` cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in `sys.modules`.
- After the module is created but before execution, the import machinery sets the import-related module attributes ("_init_module_atrs" in the pseudo-code example above), as summarized in a later section.
- Module execution is the key moment of loading in which the module's namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to `exec_module()` may not be the one returned at the end of import [2].

*Changed in version 3.4:* The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the `importlib.abc.Loader.load_module()` method.
5.4.1. Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module’s code in the module’s global name space (`module.__dict__`).
- If the loader cannot execute the module, it should raise an `ImportError`, although any other exception raised during `exec_module()` will be propagated.

In many cases, the finder and loader can be the same object; in such cases the `find_spec()` method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a `create_module()` method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

*New in version 3.4:* The `create_module()` method of loaders.

*Changed in version 3.4:* The `load_module()` method was replaced
by `exec_module()` and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.
- The module must exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove only the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

*Changed in version 3.5:* A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

*Changed in version 3.6:* An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.
5.4.2. Submodules

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module's namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let's say you have the following directory structure:

```
spam/
  __init__.py
  foo.py
  bar.py
```

and `spam/__init__.py` has the following lines in it:

```
from .foo import Foo
from .bar import Bar
```

then executing the following puts a name binding to `foo` and `bar` in the `spam` module:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.bar
<module 'spam.bar' from '/tmp/imports/spam/bar.py'>
```

Given Python’s familiar name binding rules this might seem surprising, but it’s actually a fundamental feature of the import system. The invariant holding is that if you have `sys.modules['spam']` and
sys.modules['spam.foo'] (as you would after the above import), the latter must appear as the foo attribute of the former.

5.4.3. Module spec

The import machinery uses a variety of information about each module during import, especially before loading. Most of the information is common to all modules. The purpose of a module’s spec is to encapsulate this import-related information on a per-module basis.

Using a spec during import allows state to be transferred between import system components, e.g. between the finder that creates the module spec and the loader that executes it. Most importantly, it allows the import machinery to perform the boilerplate operations of loading, whereas without a module spec the loader had that responsibility.

The module’s spec is exposed as the __spec__ attribute on a module object. See ModuleSpec for details on the contents of the module spec.

New in version 3.4.

5.4.4. Import-related module attributes

The import machinery fills in these attributes on each module object during loading, based on the module’s spec, before the loader executes the module.

__name__

The __name__ attribute must be set to the fully-qualified name of the module. This name is used to uniquely identify the module in the import system.
__loader__
The __loader__ attribute must be set to the loader object that the import machinery used when loading the module. This is mostly for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

__package__
The module's __package__ attribute must be set. Its value must be a string, but it can be the same value as its __name__. When the module is a package, its __package__ value should be set to its __name__. When the module is not a package, __package__ should be set to the empty string for top-level modules, or for submodules, to the parent package's name. See PEP 366 for further details.

This attribute is used instead of __name__ to calculate explicit relative imports for main modules, as defined in PEP 366. It is expected to have the same value as __spec__.parent.

*Changed in version 3.6*: The value of __package__ is expected to be the same as __spec__.parent.

__spec__
The __spec__ attribute must be set to the module spec that was used when importing the module. Setting __spec__ appropriately applies equally to modules initialized during interpreter startup. The one exception is __main__, where __spec__ is set to None in some cases.

When __package__ is not defined, __spec__.parent is used as a fallback.
New in version 3.4.

Changed in version 3.6: \_spec\_.parent is used as a fallback when \_package\_ is not defined.

\_path\_

If the module is a package (either regular or namespace), the module object's \_path\_ attribute must be set. The value must be iterable, but may be empty if \_path\_ has no further significance. If \_path\_ is not empty, it must produce strings when iterated over. More details on the semantics of \_path\_ are given below.

Non-package modules should not have a \_path\_ attribute.

\_file\_

\_cached\_

\_file\_ is optional. If set, this attribute's value must be a string. The import system may opt to leave \_file\_ unset if it has no semantic meaning (e.g. a module loaded from a database).

If \_file\_ is set, it may also be appropriate to set the \_cached\_ attribute which is the path to any compiled version of the code (e.g. byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file would exist (see PEP 3147).

It is also appropriate to set \_cached\_ when \_file\_ is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of \_file\_ and/or \_cached\_. So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.
5.4.5. module.__path__

By definition, if a module has a `__path__` attribute, it is a package.

A package's `__path__` attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as `sys.path`, i.e. providing a list of locations to search for modules during import. However, `__path__` is typically much more constrained than `sys.path`.

`__path__` must be an iterable of strings, but it may be empty. The same rules used for `sys.path` also apply to a package's `__path__`, and `sys.path_hooks` (described below) are consulted when traversing a package's `__path__`.

A package's `__init__.py` file may set or alter the package's `__path__` attribute, and this was typically the way namespace packages were implemented prior to PEP 420. With the adoption of PEP 420, namespace packages no longer need to supply `__init__.py` files containing only `__path__` manipulation code; the import machinery automatically sets `__path__` correctly for the namespace package.

5.4.6. Module reprs

By default, all modules have a usable repr, however depending on the attributes set above, and in the module's spec, you can more explicitly control the repr of module objects.

If the module has a spec (`__spec__`), the import machinery will try to generate a repr from it. If that fails or there is no spec, the import
system will craft a default repr using whatever information is available on the module. It will try to use the `module.__name__`, `module.__file__`, and `module.__loader__` as input into the repr, with defaults for whatever information is missing.

Here are the exact rules used:

- If the module has a `__spec__` attribute, the information in the spec is used to generate the repr. The “name”, “loader”, “origin”, and “has_location” attributes are consulted.
- If the module has a `__file__` attribute, this is used as part of the module’s repr.
- If the module has no `__file__` but does have a `__loader__` that is not `None`, then the loader’s repr is used as part of the module’s repr.
- Otherwise, just use the module’s `__name__` in the repr.

*Changed in version 3.4:* Use of `loader.module_repr()` has been deprecated and the module spec is now used by the import machinery to generate a module repr.

For backward compatibility with Python 3.3, the module repr will be generated by calling the loader’s `module_repr()` method, if defined, before trying either approach described above. However, the method is deprecated.

### 5.4.7. Cached bytecode invalidation

Before Python loads cached bytecode from `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source’s last-modified timestamp and size in the cache file when writing it. At runtime, the import system
then validates the cache file by checking the stored metadata in the cache file against at source's metadata.

Python also supports “hash-based” cache files, which store a hash of the source file's contents rather than its metadata. There are two variants of hash-based .pyc files: checked and unchecked. For checked hash-based .pyc files, Python validates the cache file by hashing the source file and comparing the resulting hash with the hash in the cache file. If a checked hash-based cache file is found to be invalid, Python regenerates it and writes a new checked hash-based cache file. For unchecked hash-based .pyc files, Python simply assumes the cache file is valid if it exists. Hash-based .pyc files validation behavior may be overridden with the --check-hash-based-pycs flag.

*Changed in version 3.7:* Added hash-based .pyc files. Previously, Python only supported timestamp-based invalidation of bytecode caches.
5.5. The Path Based Finder

As mentioned previously, Python comes with several default meta path finders. One of these, called the path based finder (PathFinder), searches an import path, which contains a list of path entries. Each path entry names a location to search for modules.

The path based finder itself doesn’t know how to import anything. Instead, it traverses the individual path entries, associating each of them with a path entry finder that knows how to handle that particular kind of path.

The default set of path entry finders implement all the semantics for finding modules on the file system, handling special file types such as Python source code (.py files), Python byte code (.pyc files) and shared libraries (e.g. .so files). When supported by the zipimport module in the standard library, the default path entry finders also handle loading all of these file types (other than shared libraries) from zipfiles.

Path entries need not be limited to file system locations. They can refer to URLs, database queries, or any other location that can be specified as a string.

The path based finder provides additional hooks and protocols so that you can extend and customize the types of searchable path entries. For example, if you wanted to support path entries as network URLs, you could write a hook that implements HTTP semantics to find modules on the web. This hook (a callable) would return a path entry finder supporting the protocol described below, which was then used to get a loader for the module from the web.
A word of warning: this section and the previous both use the term finder, distinguishing between them by using the terms meta path finder and path entry finder. These two types of finders are very similar, support similar protocols, and function in similar ways during the import process, but it's important to keep in mind that they are subtly different. In particular, meta path finders operate at the beginning of the import process, as keyed off the sys.meta_path traversal.

By contrast, path entry finders are in a sense an implementation detail of the path based finder, and in fact, if the path based finder were to be removed from sys.meta_path, none of the path entry finder semantics would be invoked.

### 5.5.1. Path entry finders

The path based finder is responsible for finding and loading Python modules and packages whose location is specified with a string path entry. Most path entries name locations in the file system, but they need not be limited to this.

As a meta path finder, the path based finder implements the find_spec() protocol previously described, however it exposes additional hooks that can be used to customize how modules are found and loaded from the import path.

Three variables are used by the path based finder, sys.path, sys.path_hooks and sys.path_importer_cache. The __path__ attributes on package objects are also used. These provide additional ways that the import machinery can be customized.

sys.path contains a list of strings providing search locations for
modules and packages. It is initialized from the PYTHONPATH environment variable and various other installation- and implementation-specific defaults. Entries in `sys.path` can name directories on the file system, zip files, and potentially other “locations” (see the `site` module) that should be searched for modules, such as URLs, or database queries. Only strings and bytes should be present on `sys.path`; all other data types are ignored. The encoding of bytes entries is determined by the individual path entry finders.

The path based finder is a meta path finder, so the import machinery begins the import path search by calling the path based finder's `find_spec()` method as described previously. When the `path` argument to `find_spec()` is given, it will be a list of string paths to traverse - typically a package's `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and `sys.path` is used.

The path based finder iterates over every entry in the search path, and for each of these, looks for an appropriate path entry finder (PathEntryFinder) for the path entry. Because this can be an expensive operation (e.g. there may be `stat()` call overheads for this search), the path based finder maintains a cache mapping path entries to path entry finders. This cache is maintained in `sys.path_importer_cache` (despite the name, this cache actually stores finder objects rather than being limited to importer objects). In this way, the expensive search for a particular path entry location's path entry finder need only be done once. User code is free to remove cache entries from `sys.path_importer_cache` forcing the path based finder to perform the path entry search again [3].

If the path entry is not present in the cache, the path based finder iterates over every callable in `sys.path_hooks`. Each of the path
**entry hooks** in this list is called with a single argument, the path entry to be searched. This callable may either return a **path entry finder** that can handle the path entry, or it may raise **ImportError**. An **ImportError** is used by the path based finder to signal that the hook cannot find a **path entry finder** for that **path entry**. The exception is ignored and import path iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise **ImportError**.

If **sys.path_hooks** iteration ends with no **path entry finder** being returned, then the path based finder’s **find_spec()** method will store **None** in **sys.path_importer_cache** (to indicate that there is no finder for this path entry) and return **None**, indicating that this **meta path finder** could not find the module.

If a **path entry finder** is returned by one of the **path entry hook** callables on **sys.path_hooks**, then the following protocol is used to ask the finder for a module spec, which is then used when loading the module.

The current working directory – denoted by an empty string – is handled slightly differently from other entries on **sys.path**. First, if the current working directory is found to not exist, no value is stored in **sys.path_importer_cache**. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for **sys.path_importer_cache** and returned by **importlib.machinery.PathFinder.find_spec()** will be the actual current working directory and not the empty string.

5.5.2. Path entry finder protocol
In order to support imports of modules and initialized packages and also to contribute portions to namespace packages, path entry finders must implement the `find_spec()` method.

`find_spec()` takes two argument, the fully qualified name of the module being imported, and the (optional) target module. `find_spec()` returns a fully populated spec for the module. This spec will always have “loader” set (with one exception).

To indicate to the import machinery that the spec represents a namespace portion, the path entry finder sets “loader” on the spec to `None` and “submodule_search_locations” to a list containing the portion.

*Changed in version 3.4:* `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Older path entry finders may implement one of these two deprecated methods instead of `find_spec()`. The methods are still respected for the sake of backward compatibility. However, if `find_spec()` is implemented on the path entry finder, the legacy methods are ignored.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace portion. When the first item (i.e. the loader) is `None`, this means that while the path entry finder does not have a loader for the named module, it knows that the path entry contributes to a namespace portion for the named module. This will almost always be the case where Python is asked to import a namespace package that has no physical presence on the file system. When a path entry finder returns `None` for the
loader, the second item of the 2-tuple return value must be a sequence, although it can be empty.

If \texttt{find\_loader()} returns a non-\texttt{None} loader value, the portion is ignored and the loader is returned from the path based finder, terminating the search through the path entries.

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional \texttt{find\_module()} method that meta path finders support. However path entry finder \texttt{find\_module()} methods are never called with a \texttt{path} argument (they are expected to record the appropriate path information from the initial call to the path hook).

The \texttt{find\_module()} method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both \texttt{find\_loader()} and \texttt{find\_module()} exist on a path entry finder, the import system will always call \texttt{find\_loader()} in preference to \texttt{find\_module()}. 
5.6. Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient. This technique may also be employed at the module level to only alter the behaviour of import statements within that module.

To selectively prevent import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.
5.7. Special considerations for __main__

The __main__ module is a special case relative to Python’s import system. As noted elsewhere, the __main__ module is directly initialized at interpreter startup, much like sys and builtins. However, unlike those two, it doesn’t strictly qualify as a built-in module. This is because the manner in which __main__ is initialized depends on the flags and other options with which the interpreter is invoked.

5.7.1. __main__.__spec__

Depending on how __main__ is initialized, __main__.__spec__ gets set appropriately or to None.

When Python is started with the -m option, __spec__ is set to the module spec of the corresponding module or package. __spec__ is also populated when the __main__ module is loaded as part of executing a directory, zipfile or other sys.path entry.

In the remaining cases __main__.__spec__ is set to None, as the code used to populate the __main__ does not correspond directly with an importable module:

- interactive prompt
- -c option
- running from stdin
- running directly from a source or bytecode file

Note that __main__.__spec__ is always None in the last case, even if the file could technically be imported directly as a module
instead. Use the `-m` switch if valid module metadata is desired in `__main__`.

Note also that even when `__main__` corresponds with an importable module and `__main__.__spec__` is set accordingly, they're still considered distinct modules. This is due to the fact that blocks guarded by `if __name__ == "__main__":` checks only execute when the module is used to populate the `__main__` namespace, and not during normal import.
5.8. Open issues

XXX It would be really nice to have a diagram.

XXX * (import_machinery.rst) how about a section devoted just to the attributes of modules and packages, perhaps expanding upon or supplanting the related entries in the data model reference page?

XXX runpy, pkgutil, et al in the library manual should all get “See Also” links at the top pointing to the new import system section.

XXX Add more explanation regarding the different ways in which __main__ is initialized?

XXX Add more info on __main__ quirks/pitfalls (i.e. copy from PEP 395).
5.9. References

The import machinery has evolved considerably since Python’s early days. The original specification for packages is still available to read, although some details have changed since the writing of that document.

The original specification for `sys.meta_path` was PEP 302, with subsequent extension in PEP 420.

**PEP 420** introduced namespace packages for Python 3.3. **PEP 420** also introduced the `find_loader()` protocol as an alternative to `find_module()`.

**PEP 366** describes the addition of the `__package__` attribute for explicit relative imports in main modules.

**PEP 328** introduced absolute and explicit relative imports and initially proposed `__name__` for semantics **PEP 366** would eventually specify for `__package__`.

**PEP 338** defines executing modules as scripts.

**PEP 451** adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

Footnotes

The importlib implementation avoids using the return value directly. Instead, it gets the module object by looking the module name up in `sys.modules`. The indirect effect of this is that an imported module may replace itself in `sys.modules`. This is implementation-specific behavior that is not guaranteed to work in other Python implementations.

In legacy code, it is possible to find instances of `imp.NullImporter` in the `sys.path_importer_cache`. It is recommended that code be changed to use `None` instead. See Porting Python code for more details.
6. Expressions

This chapter explains the meaning of the elements of expressions in Python.

**Syntax Notes:** In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

\[
\text{name} ::= \text{othername}
\]

and no semantics are given, the semantics of this form of name are the same as for othername.
6.1. Arithmetic conversions

When a description of an arithmetic operator below uses the phrase “the numeric arguments are converted to a common type,” this means that the operator implementation for built-in types works as follows:

- If either argument is a complex number, the other is converted to complex;
- otherwise, if either argument is a floating point number, the other is converted to floating point;
- otherwise, both must be integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string as a left argument to the ‘%’ operator). Extensions must define their own conversion behavior.
6.2. Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

\[
\begin{align*}
\text{atom} & \ ::= \ \text{identifier} \ | \ \text{literal} \ | \ \text{enclosure} \\
\text{enclosure} & \ ::= \ \text{parenth_form} \ | \ \text{list_display} \ | \ \text{dict_display} \\
& \quad | \ \text{generator_expression} \ | \ \text{yield_atom}
\end{align*}
\]

6.2.1. Identifiers (Names)

An identifier occurring as an atom is a name. See section Identifiers and keywords for lexical definition and section Naming and binding for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a NameError exception.

Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a private name of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name, with leading underscores removed and a single underscore inserted, in front of the name. For example, the identifier __spam occurring in a class named Ham will be transformed to _Ham__spam. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is
extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

6.2.2. Literals

Python supports string and bytes literals and various numeric literals:

\[
\text{literal ::= \ stringliteral} \mid \text{bytesliteral} \\
\mid \text{integer} \mid \text{floatnumber} \mid \text{imagnumber}
\]

Evaluation of a literal yields an object of the given type (string, bytes, integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section Literals for details.

All literals correspond to immutable data types, and hence the object’s identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

6.2.3. Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

\[
\text{parenth_form ::= "} [\text{starred_expression}] \text{"}
\]

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression
An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses are required — allowing unparenthesized “nothing” in expressions would cause ambiguities and allow common typos to pass uncaught.

6.2.4. Displays for lists, sets and dictionaries

For constructing a list, a set or a dictionary Python provides special syntax called “displays”, each of them in two flavors:

- either the container contents are listed explicitly, or
- they are computed via a set of looping and filtering instructions, called a comprehension.

Common syntax elements for comprehensions are:

- `comprehension ::= expression comp_for`
- `comp_for ::= ["async"] "for" target_list "in" comp_iter`
- `comp_iter ::= comp_for | comp_if`
- `comp_if ::= "if" expression_nocond [comp_iter]`

The comprehension consists of a single expression followed by at least one for clause and zero or more for or if clauses. In this case, the elements of the new container are those that would be produced by considering each of the for or if clauses a block, nesting from left to right, and evaluating the expression to produce an
element each time the innermost block is reached.

However, aside from the iterable expression in the leftmost for clause, the comprehension is executed in a separate implicitly nested scope. This ensures that names assigned to in the target list don’t “leak” into the enclosing scope.

The iterable expression in the leftmost for clause is evaluated directly in the enclosing scope and then passed as an argument to the implicitly nested scope. Subsequent for clauses and any filter condition in the leftmost for clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: \([x*y \text{ for } x \text{ in } \text{range}(10) \text{ for } y \text{ in } \text{range}(x, x+10)]\).

To ensure the comprehension always results in a container of the appropriate type, \texttt{yield} and \texttt{yield from} expressions are prohibited in the implicitly nested scope (in Python 3.7, such expressions emit \texttt{DeprecationWarning} when compiled, in Python 3.8+ they will emit \texttt{SyntaxError}).

Since Python 3.6, in an \texttt{async def} function, an \texttt{async} for clause may be used to iterate over a \texttt{asynchronous iterator}. A comprehension in an \texttt{async def} function may consist of either a for or async for clause following the leading expression, may contain additional for or async for clauses, and may also use \texttt{await} expressions. If a comprehension contains either async for clauses or await expressions it is called an \texttt{asynchronous comprehension}. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also \texttt{PEP 530}.

\textit{New in version 3.6}: Asynchronous comprehensions were introduced.
Deprecated since version 3.7: `yield` and `yield from` deprecated in the implicitly nested scope.

6.2.5. List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display ::= "[" [starred_list | comprehension] "]"
```

A list display yields a new list object, the contents being specified by either a list of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a comprehension is supplied, the list is constructed from the elements resulting from the comprehension.

6.2.6. Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

```
set_display ::= "{" (starred_list | comprehension) "}"
```

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.
An empty set cannot be constructed with `{}`; this literal constructs an empty dictionary.

### 6.2.7. Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```plaintext
dict_display ::= "{" [key_datum_list | dict_comprehension]"
key_datum_list ::= key_datum ("," key_datum)*
key_datum ::= expression "_:" expression |
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of key/datum pairs is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum. This means that you can specify the same key multiple times in the key/datum list, and the final dictionary's value for that key will be the last one given.

A double asterisk `**` denotes dictionary unpacking. Its operand must be a mapping. Each mapping item is added to the new dictionary. Later values replace values already set by earlier key/datum pairs and earlier dictionary unpackings.

*New in version 3.5:* Unpacking into dictionary displays, originally proposed by PEP 448.

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual “for” and “if” clauses. When the comprehension is run, the resulting
key and value elements are inserted in the new dictionary in the order they are produced.

Restrictions on the types of the key values are listed earlier in section The standard type hierarchy. (To summarize, the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

6.2.8. Generator expressions

A generator expression is a compact generator notation in parentheses:

```python
generator_expression ::= "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for the generator object (in the same fashion as normal generators). However, the iterable expression in the leftmost `for` clause is immediately evaluated, so that an error produced by it will be emitted at the point where the generator expression is defined, rather than at the point where the first value is retrieved. Subsequent `for` clauses and any filter condition in the leftmost `for` clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `(x*y for x in range(10) for y in range(x, x+10))`. 
The parentheses can be omitted on calls with only one argument. See section \texttt{Calls} for details.

To avoid interfering with the expected operation of the generator expression itself, \texttt{yield} and \texttt{yield from} expressions are prohibited in the implicitly defined generator (in Python 3.7, such expressions emit \texttt{DeprecationWarning} when compiled, in Python 3.8+ they will emit \texttt{SyntaxError}).

If a generator expression contains either \texttt{async} for clauses or \texttt{await} expressions it is called an \texttt{asynchronous generator expression}. An asynchronous generator expression returns a new asynchronous generator object, which is an asynchronous iterator (see \texttt{Asynchronous Iterators}).

\textit{New in version 3.6}: Asynchronous generator expressions were introduced.

\textit{Changed in version 3.7}: Prior to Python 3.7, asynchronous generator expressions could only appear in \texttt{async def} coroutines. Starting with 3.7, any function can use asynchronous generator expressions.

\textit{Deprecated since version 3.7}: \texttt{yield} and \texttt{yield from} deprecated in the implicitly nested scope.

\textbf{6.2.9. Yield expressions}

\begin{verbatim}
yield_atom ::= "(" yield_expression ")"
yield_expression ::= "yield" [expression_list | "from"]
\end{verbatim}

The yield expression is used when defining a \texttt{generator} function or an \texttt{asynchronous generator} function and thus can only be used in the
body of a function definition. Using a yield expression in a function’s body causes that function to be a generator, and using it in an `async def` function’s body causes that coroutine function to be an asynchronous generator. For example:

```python
def gen():  # defines a generator function
    yield 123
async def agen(): # defines an asynchronous generator function
    yield 123
```

Due to their side effects on the containing scope, `yield` expressions are not permitted as part of the implicitly defined scopes used to implement comprehensions and generator expressions (in Python 3.7, such expressions emit `DeprecationWarning` when compiled, in Python 3.8+ they will emit `SyntaxError`).

*Deprecated since version 3.7:* Yield expressions deprecated in the implicitly nested scopes used to implement comprehensions and generator expressions.

Generator functions are described below, while asynchronous generator functions are described separately in section [Asynchronous generator functions](#).

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator’s methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of `expression_list` to the generator's caller. By suspended, we mean that all local state is retained, including the current bindings of local
variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If __next__() is used (typically via either a for or the next() builtin) then the result is None. Otherwise, if send() is used, then the result will be the value passed in to that method.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where the execution should continue after it yields; the control is always transferred to the generator’s caller.

Yield expressions are allowed anywhere in a try construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator’s close() method will be called, allowing any pending finally clauses to execute.

When yield from <expr> is used, it treats the supplied expression as a subiterator. All values produced by that subiterator are passed directly to the caller of the current generator’s methods. Any values passed in with send() and any exceptions passed in with throw() are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then send() will raise AttributeError or TypeError, while throw() will just raise the passed in exception immediately.

When the underlying iterator is complete, the value attribute of the
raised :obj:`StopIteration` instance becomes the value of the yield expression. It can be either set explicitly when raising :obj:`StopIteration`, or automatically when the sub-iterator is a generator (by returning a value from the sub-generator).

Changed in version 3.3: Added :obj:`yield from <expr>` to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

**See also:**

**PEP 255 - Simple Generators**
The proposal for adding generators and the :obj:`yield` statement to Python.

**PEP 342 - Coroutines via Enhanced Generators**
The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

**PEP 380 - Syntax for Delegating to a Subgenerator**
The proposal to introduce the :obj:`yield_from` syntax, making delegation to sub-generators easy.

**PEP 525 - Asynchronous Generators**
The proposal that expanded on **PEP 492** by adding generator capabilities to coroutine functions.

6.2.9.1. Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the
generator is already executing raises a `ValueError` exception.

generator.__next__()

Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a `__next__()` method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the `expression_list` is returned to `__next__()`'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

This method is normally called implicitly, e.g. by a `for` loop, or by the built-in `next()` function.

generator.send(value)

Resumes the execution and “sends” a value into the generator function. The `value` argument becomes the result of the current yield expression. The `send()` method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When `send()` is called to start the generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

generator.throw(type[, value[, traceback]])

Raises an exception of type `type` at the point where the generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different
exception, then that exception propagates to the caller.

generator.close()

Raises a GeneratorExit at the point where the generator function was paused. If the generator function then exits gracefully, is already closed, or raises GeneratorExit (by not catching the exception), close returns to its caller. If the generator yields a value, a RuntimeError is raised. If the generator raises any other exception, it is propagated to the caller. close() does nothing if the generator has already exited due to an exception or normal exit.

6.2.9.2. Examples

Here is a simple example that demonstrates the behavior of generators and generator functions:

```python
>>> def echo(value=\None):
...     print("Execution starts when 'next()' is called.
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.
... >>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
```
For examples using `yield from`, see PEP 380: Syntax for Delegating to a Subgenerator in “What’s New in Python.”

### 6.2.9.3. Asynchronous generator functions

The presence of a yield expression in a function or method defined using `async def` further defines the function as an asynchronous generator function.

When an asynchronous generator function is called, it returns an asynchronous iterator known as an asynchronous generator object. That object then controls the execution of the generator function. An asynchronous generator object is typically used in an `async for` statement in a coroutine function analogously to how a generator object would be used in a `for` statement.

Calling one of the asynchronous generator’s methods returns an `awaitable` object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of `expression_list` to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the
asynchronous generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If \_\_anext\_\_() is used then the result is None. Otherwise, if asend() is used, then the result will be the value passed in to that method.

In an asynchronous generator function, yield expressions are allowed anywhere in a try construct. However, if an asynchronous generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), then a yield expression within a try construct could result in a failure to execute pending finally clauses. In this case, it is the responsibility of the event loop or scheduler running the asynchronous generator to call the asynchronous generator-iterator's aclose() method and run the resulting coroutine object, thus allowing any pending finally clauses to execute.

To take care of finalization, an event loop should define a finalizer function which takes an asynchronous generator-iterator and presumably calls aclose() and executes the coroutine. This finalizer may be registered by calling sys.set_asyncgen_hooks(). When first iterated over, an asynchronous generator-iterator will store the registered finalizer to be called upon finalization. For a reference example of a finalizer method see the implementation of asyncio.Loop.shutdown_asyncgens in Lib/asyncio/base_events.py.

The expression yield from <expr> is a syntax error when used in an asynchronous generator function.

6.2.9.4. Asynchronous generator-iterator methods
This subsection describes the methods of an asynchronous generator iterator, which are used to control the execution of a generator function.

**coroutine agen.**\[\texttt{\_\_anext\_}()\]

Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with a \[\texttt{\_\_anext\_}()\] method, the current yield expression always evaluates to \texttt{None} in the returned awaitable, which when run will continue to the next yield expression. The value of the \texttt{expression_list} of the yield expression is the value of the \texttt{StopIteration} exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises an \texttt{StopAsyncIteration} exception, signalling that the asynchronous iteration has completed.

This method is normally called implicitly by a \texttt{async for} loop.

**coroutine agen.**\[\texttt{asend(value)}\]

Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the \texttt{send()} method for a generator, this “sends” a value into the asynchronous generator function, and the \texttt{value} argument becomes the result of the current yield expression. The awaitable returned by the \texttt{asend()} method will return the next value yielded by the generator as the value of the raised \texttt{StopIteration}, or raises \texttt{StopAsyncIteration} if the asynchronous generator exits without yielding another value. When \texttt{asend()} is called to start the asynchronous generator, it must be called with \texttt{None} as the argument, because there is no yield expression that could receive the value.
coroutine agen. **athrow**(type[, value[, traceback]])

Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, an `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

coroutine agen. **aclose**()

Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a `StopAsyncIteration` exception. If the asynchronous generator yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to `aclose()` will return an awaitable that does nothing.
6.3. Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary ::= atom | attributeref | subscription
```

6.3.1. Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref ::= primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. This production can be customized by overriding the `__getattr__()` method. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

6.3.2. Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription ::= primary "[" expression_list "]"
```

The primary must evaluate to an object that supports subscription (lists
or dictionaries for example). User-defined objects can support
subscription by defining a `__getitem__() method.

For built-in objects, there are two types of objects that support
subscription:

If the primary is a mapping, the expression list must evaluate to an
object whose value is one of the keys of the mapping, and the
subscription selects the value in the mapping that corresponds to that
key. (The expression list is a tuple except if it has exactly one item.)

If the primary is a sequence, the expression list must evaluate to an
integer or a slice (as discussed in the following section).

The formal syntax makes no special provision for negative indices in
sequences; however, built-in sequences all provide a
`__getitem__() method that interprets negative indices by adding
the length of the sequence to the index (so that `x[-1]` selects the last
item of `x`). The resulting value must be a nonnegative integer less than
the number of items in the sequence, and the subscription selects the
item whose index is that value (counting from zero). Since the support
for negative indices and slicing occurs in the object's
`__getitem__() method, subclasses overriding this method will need
to explicitly add that support.

A string's items are characters. A character is not a separate data type
but a string of exactly one character.

6.3.3. Slicings

A slicing selects a range of items in a sequence object (e.g., a string,
tuple or list). Slicings may be used as expressions or as targets in
assignment or `del` statements. The syntax for a slicing:

```
<table>
<thead>
<tr>
<th>Slicing</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>slicing ::= primary &quot;[&quot; slice_list &quot;]&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>slice_list ::= slice_item (&quot;,&quot; slice_item)* [&quot;&quot;,&quot;)</code></td>
<td></td>
</tr>
<tr>
<td>`slice_item ::= expression</td>
<td>proper_slice`</td>
</tr>
<tr>
<td><code>proper_slice ::= [lower_bound] &quot;:&quot; [upper_bound] [stride]</code></td>
<td></td>
</tr>
<tr>
<td><code>lower_bound ::= expression</code></td>
<td></td>
</tr>
<tr>
<td><code>upper_bound ::= expression</code></td>
<td></td>
</tr>
<tr>
<td><code>stride ::= expression</code></td>
<td></td>
</tr>
</tbody>
</table>
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice).

The semantics for a slicing are as follows. The primary is indexed (using the same `__getitem__()` method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section `The standard type hierarchy`) whose start, stop and step attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

### 6.3.4. Calls

A call calls a callable object (e.g., a `function`) with a possibly empty
series of arguments:

```plaintext
call ::= primary "(" [argument_list] ")"
argument_list ::= positional_arguments [","
                  | starred_and_keywords [","
                  | keywords_arguments
positional_arguments ::= ["*"] expression ("," ["*"]
starred_and_keywords ::= ("*" expression | keyword_item
                        ("," "*" expression | ",",
                        | keywords_arguments
keywords_arguments ::= (keyword_item | "**" expression
                      ("," keyword_item | ",""
keyword_item ::= identifier "=" expression
```

An optional trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a `__call__()` method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section Function definitions for the syntax of formal parameter lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a TypeError exception is raised. Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is None, it fills the slot). When all arguments
have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don’t specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

**CPython implementation detail:** An implementation may provide built-in functions whose positional parameters do not have names, even if they are ‘named’ for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use `PyArg_ParseTuple()` to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression`
must evaluate to an iterable. Elements from these iterables are treated as if they were additional positional arguments. For the call \( f(x_1, x_2, \ast y, x_3, x_4) \), if \( y \) evaluates to a sequence \( y_1, \ldots, y_M \), this is equivalent to a call with \( M+4 \) positional arguments \( x_1, x_2, y_1, \ldots, y_M, x_3, x_4 \).

A consequence of this is that although the \( \ast \text{expression} \) syntax may appear after explicit keyword arguments, it is processed before the keyword arguments (and any \( \ast\ast \text{expression} \) arguments – see below). So:

```python
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the \( \ast \text{expression} \) syntax to be used in the same call, so in practice this confusion does not arise.

If the syntax \( \ast\ast \text{expression} \) appears in the function call, expression must evaluate to a \texttt{mapping}, the contents of which are treated as additional keyword arguments. If a keyword is already present (as an explicit keyword argument, or from another unpacking), a \texttt{TypeError} exception is raised.
Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names.

Changed in version 3.5: Function calls accept any number of `*` and `**` unpackings, positional arguments may follow iterable unpackings (`*`), and keyword arguments may follow dictionary unpackings (`**`). Originally proposed by PEP 448.

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function:

The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section Function definitions. When the code block executes a `return` statement, this specifies the return value of the function call.

a built-in function or method:

The result is up to the interpreter; see Built-in Functions for the descriptions of built-in functions and methods.

a class object:

A new instance of that class is returned.

a class instance method:

The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.
a class instance:

The class must define a `__call__()` method; the effect is then the same as if that method was called.
6.4. Await expression

Suspend the execution of coroutine on an awaitable object. Can only be used inside a coroutine function.

```
await_expr ::= "await" primary
```

New in version 3.5.
6.5. The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power ::= (await_expr | primary) ["***" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): `-1**2` results in `-1`.

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type, and the result is of that type.

For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`.

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a complex number. (In earlier versions it raised a `ValueError`.)
6.6. Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

\[
\text{u}_\text{expr} ::= \text{power} \mid "-" \text{u}_\text{expr} \mid "+" \text{u}_\text{expr} \mid "\sim" \text{u}_\text{expr}
\]

The unary \(-\) (minus) operator yields the negation of its numeric argument.

The unary \(+\) (plus) operator yields its numeric argument unchanged.

The unary \(~\) (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of \(x\) is defined as \(-(x+1)\). It only applies to integral numbers.

In all three cases, if the argument does not have the proper type, a \texttt{TypeError} exception is raised.
6.7. Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

\[
\begin{align*}
m_{\text{expr}} & ::= u_{\text{expr}} \mid m_{\text{expr}} \ast u_{\text{expr}} \mid m_{\text{expr}} \od m_{\text{expr}} \div u_{\text{expr}} \\
& \mid m_{\text{expr}} \mod u_{\text{expr}} \\
a_{\text{expr}} & ::= m_{\text{expr}} \mid a_{\text{expr}} + m_{\text{expr}} \mid a_{\text{expr}} - m_{\text{expr}} \\
\end{align*}
\]

The \( \ast \) (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The \( \od \) (at) operator is intended to be used for matrix multiplication. No built-in Python types implement this operator.

*New in version 3.5.*

The \( \div \) (division) and \( \od \) (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Division of integers yields a float, while floor division of integers results in an integer; the result is that of mathematical division with the ‘floor’ function applied to the result. Division by zero raises the \texttt{ZeroDivisionError} exception.

The \( \mod \) (modulo) operator yields the remainder from the division of the
first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the \texttt{ZeroDivisionError} exception. The arguments may be floating point numbers, e.g., \texttt{3.14\%0.7} equals \texttt{0.34} (since \texttt{3.14} equals \texttt{4\*0.7 + 0.34}). The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand \cite{1}.

The floor division and modulo operators are connected by the following identity: \texttt{x == (x//y)*y + (x\%y)}. Floor division and modulo are also connected with the built-in function \texttt{divmod()}: \texttt{divmod(x, y) == (x//y, x\%y)}. \cite{2}.

In addition to performing the modulo operation on numbers, the \% operator is also overloaded by string objects to perform old-style string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section printf-style String Formatting.

The floor division operator, the modulo operator, and the \texttt{divmod()} function are not defined for complex numbers. Instead, convert to a floating point number using the \texttt{abs()} function if appropriate.

The \texttt{+} (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The \texttt{-} (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.
6.8. Shifting operations

The shifting operations have lower priority than the arithmetic operations:

\[
\text{shift_expr} ::= \text{a_expr} \mid \text{shift_expr} ("<<" | ">>") \text{a_expr}
\]

These operators accept integers as arguments. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by \( n \) bits is defined as floor division by \( \text{pow}(2, n) \). A left shift by \( n \) bits is defined as multiplication with \( \text{pow}(2, n) \).
6.9. Binary bitwise operations

Each of the three bitwise operations has a different priority level:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>and_expr ::= shift_expr</td>
<td>and_expr &quot;&amp;&quot; shift_expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor_expr ::= and_expr</td>
<td>xor_expr &quot;^&quot; and_expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or_expr ::= xor_expr</td>
<td>or_expr &quot;</td>
<td>&quot; xor_expr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The & operator yields the bitwise AND of its arguments, which must be integers.

The ^ operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers.

The | operator yields the bitwise (inclusive) OR of its arguments, which must be integers.
6.10. Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison ::= or_expr (comp_operator or_expr)*
comp_operator ::= "<" | ">" | ">=": | "=" | "<=" |
                | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: True or False.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if `a, b, c, ..., y, z` are expressions and `op1, op2, ..., opN` are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b` and `b op2 c` and ... `y opN z`, except that each expression is evaluated at most once.

Note that `a op1 b op2 c` doesn’t imply any kind of comparison between `a` and `c`, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

6.10.1. Value comparisons

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the values of two objects. The objects do not need to have the same type.
Chapter Objects, values and types states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python: For example, there is no canonical access method for an object's value. Also, there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Because all types are (direct or indirect) subtypes of object, they inherit the default comparison behavior from object. Types can customize their comparison behavior by implementing rich comparison methods like __lt__( ), described in Basic customization.

The default behavior for equality comparison (== and !=) is based on the identity of the objects. Hence, equality comparison of instances with the same identity results in equality, and equality comparison of instances with different identities results in inequality. A motivation for this default behavior is the desire that all objects should be reflexive (i.e. x is y implies x == y).

A default order comparison (,<, >, <=, and >=) is not provided; an attempt raises TypeError. A motivation for this default behavior is the lack of a similar invariant as for equality.

The behavior of the default equality comparison, that instances with different identities are always unequal, may be in contrast to what types will need that have a sensible definition of object value and value-based equality. Such types will need to customize their comparison behavior, and in fact, a number of built-in types have done that.
The following list describes the comparison behavior of the most important built-in types.

- Numbers of built-in numeric types (Numeric Types — int, float, complex) and of the standard library types fractions.Fraction and decimal.Decimal can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

The not-a-number values float('NaN') and decimal.Decimal('NaN') are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number values are not equal to themselves. For example, if \( x = \text{float('NaN')} \), \( 3 < x \), \( x < 3 \), \( x == x \), \( x != x \) are all false. This behavior is compliant with IEEE 754.

- Binary sequences (instances of bytes or bytearray) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.

- Strings (instances of str) compare lexicographically using the numerical Unicode code points (the result of the built-in function ord()) of their characters. [3]

Strings and binary sequences cannot be directly compared.

- Sequences (instances of tuple, list, or range) can be compared only within each of their types, with the restriction that ranges do not support order comparison. Equality comparison across these types results in inequality, and ordering comparison
across these types raises `TypeError`.

Sequences compare lexicographically using comparison of corresponding elements, whereby reflexivity of the elements is enforced.

In enforcing reflexivity of elements, the comparison of collections assumes that for a collection element `x`, `x == x` is always true. Based on that assumption, element identity is compared first, and element comparison is performed only for distinct elements. This approach yields the same result as a strict element comparison would, if the compared elements are reflexive. For non-reflexive elements, the result is different than for strict element comparison, and may be surprising: The non-reflexive not-a-number values for example result in the following comparison behavior when used in a list:

```python
>>> nan = float('NaN')
>>> nan is nan
True
>>> nan == nan
False                  <-- the defined non-reflexive behaviour
>>> [nan] == [nan]
True                   <-- list enforces reflexivity
```

Lexicographical comparison between built-in collections works as follows:

- For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).
Collections that support order comparison are ordered the same as their first unequal elements (for example, \([1,2,x] \leq [1,2,y]\) has the same value as \(x \leq y\)). If a corresponding element does not exist, the shorter collection is ordered first (for example, \([1,2] < [1,2,3]\) is true).

- Mappings (instances of `dict`) compare equal if and only if they have equal \((key, value)\) pairs. Equality comparison of the keys and values enforces reflexivity.

Order comparisons \((<, >, \leq, \text{and} \geq)\) raise `TypeError`.

- Sets (instances of `set` or `frozenset`) can be compared within and across their types.

  They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets \(\{1,2\}\) and \(\{2,3\}\) are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, \(\text{min()}, \text{max()}, \text{and} \text{sorted()}\) produce undefined results given a list of sets as inputs).

  Comparison of sets enforces reflexivity of its elements.

- Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible:

- Equality comparison should be reflexive. In other words, identical objects should compare equal:
\[
\text{x \ is \ y \ implies \ x == y}
\]

- Comparison should be symmetric. In other words, the following expressions should have the same result:

\[
x == y \text{ and } y == x
\]

\[
x != y \text{ and } y != x
\]

\[
x < y \text{ and } y > x
\]

\[
x <= y \text{ and } y >= x
\]

- Comparison should be transitive. The following (non-exhaustive) examples illustrate that:

\[
x > y \text{ and } y > z \text{ implies } x > z
\]

\[
x < y \text{ and } y <= z \text{ implies } x < z
\]

- Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result:

\[
x == y \text{ and } not x != y
\]

\[
x < y \text{ and } not x >= y \text{ (for total ordering)}
\]

\[
x > y \text{ and } not x <= y \text{ (for total ordering)}
\]

The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the \texttt{total\_ordering()} decorator.

- The \texttt{hash()} result should be consistent with equality. Objects that are equal should either have the same hash value, or be
marked as unhashable.

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2. Membership test operations

The operators `in` and `not in` test for membership. `x in s` evaluates to `True` if `x` is a member of `s`, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which `in` tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is `True` if and only if `x` is a substring of `y`. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `' ' in "abc"` will return `True`.

For user-defined classes which define the `__contains__()` method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define `__contains__()` but do define `__iter__()`, `x in y` is `True` if some value `z` with `x == z` is produced while iterating over `y`. If an exception is raised during the iteration, it is as if `in` raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines `__getitem__()`, `x in y` is `True` if and only if there is a non-negative integer index `i` such that `x == y[i]`, and all lower integer
indices do not raise \texttt{IndexError} exception. (If any other exception is raised, it is as if \texttt{in} raised that exception).

The operator \texttt{not in} is defined to have the inverse true value of \texttt{in}.

6.10.3. Identity comparisons

The operators \texttt{is} and \texttt{is not} test for object identity: \texttt{x is y} is true if and only if \texttt{x} and \texttt{y} are the same object. Object identity is determined using the \texttt{id()} function. \texttt{x is not y} yields the inverse truth value. \cite{4}
6.11. Boolean operations

```
orf_test ::= and_test | or_test "or" and_test
and_test ::= not_test | and_test "and" not_test
not_test ::= comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: False, None, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a `__bool__()` method.

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates `x`; if `x` is false, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

The expression `x or y` first evaluates `x`; if `x` is true, its value is returned; otherwise, `y` is evaluated and the resulting value is returned.

Note that neither `and` nor `or` restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to create a new value, it returns a boolean value regardless of the type of its argument (for example, `not 'foo'` produces `False` rather than `'''`).
Conditional expressions (sometimes called a “ternary operator”) have the lowest priority of all Python operations.

The expression `x if C else y` first evaluates the condition, `C` rather than `x`. If `C` is true, `x` is evaluated and its value is returned; otherwise, `y` is evaluated and its value is returned.

See PEP 308 for more details about conditional expressions.
Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with:

```python
def <lambda>(parameters):
    return expression
```

See section *Function definitions* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.
6.14. Expression lists

```
expression_list ::= expression ("," expression)*
stared_list ::= starred_item ("," stared_item)*
stared_expression ::= expression | (stared_item ",")*
stared_item ::= expression | "+" or_expr
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an iterable. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

*New in version 3.5:* Iterable unpacking in expression lists, originally proposed by [PEP 448](https://www.python.org/dev/peps/pep-0448/).

The trailing comma is required only to create a single tuple (a.k.a. a singleton); it is optional in all other cases. A single expression without a trailing comma doesn’t create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)
6.15. Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```python
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```
6.16. Operator precedence

The following table summarizes the operator precedence in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the Comparisons section.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>Lambda expression</td>
</tr>
<tr>
<td>if – else</td>
<td>Conditional expression</td>
</tr>
<tr>
<td>or</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>and</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>not x</td>
<td>Boolean NOT</td>
</tr>
<tr>
<td>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</td>
<td>Comparisons, including membership tests and identity tests</td>
</tr>
<tr>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>Shifts</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td></td>
<td>Multiplication, matrix</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>*, @, /, //, %</td>
<td>multiplication, division, floor division, remainder [5]</td>
</tr>
<tr>
<td>+x, -x, ~x</td>
<td>Positive, negative, bitwise NOT</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation [6]</td>
</tr>
<tr>
<td>await x</td>
<td>Await expression</td>
</tr>
<tr>
<td>x[index], x[index:index], x(arguments...), x.attribute</td>
<td>Subscription, slicing, call, attribute reference</td>
</tr>
<tr>
<td>(expressions...), [expressions...], {key: value...}, {expressions...}</td>
<td>Binding or tuple display, list display, dictionary display, set display</td>
</tr>
</tbody>
</table>

**Footnotes**

While abs(x%y) < abs(y) is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that -1e-100 % 1e100 have the same sign as 1e100, the computed result is -1e-100 + 1e100, which is numerically exactly equal to 1e100. The function `math.fmod()` returns a result whose sign matches the sign of the first argument instead, and so returns -1e-100 in this case. Which approach is more appropriate depends on the application.

If x is very close to an exact integer multiple of y, it's possible for x//y to be one larger than (x-x%y)//y due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x, y)[0] * y + x % y` be very close to x.

The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. “LATIN CAPITAL
LETTER A”). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character “LATIN CAPITAL LETTER C WITH CEDILLA” can be represented as a single precomposed character at code position U+00C7, or as a sequence of a base character at code position U+0043 (LATIN CAPITAL LETTER C), followed by a combining character at code position U+0327 (COMBINING CEDILLA).

The comparison operators on strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, "\u00C7" == "\u0043\u0327" is False, even though both strings represent the same abstract character “LATIN CAPITAL LETTER C WITH CEDILLA”.

To compare strings at the level of abstract characters (that is, in a way intuitive to humans), use unicodedata.normalize().

Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the is operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

The % operator is also used for string formatting; the same precedence applies.

The power operator ** binds less tightly than an arithmetic or bitwise unary operator on its right, that is, 2**-1 is 0.5.
7. Simple statements

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt ::= expression_stmt
               | assert_stmt
               | assignment_stmt
               | augmented_assignment_stmt
               | annotated_assignment_stmt
               | pass_stmt
               | del_stmt
               | return_stmt
               | yield_stmt
               | raise_stmt
               | break_stmt
               | continue_stmt
               | import_stmt
               | future_stmt
               | global_stmt
               | nonlocal_stmt
```
7.1. Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value None). Other uses of expression statements are allowed and occasionally useful.

The syntax for an expression statement is:

```
expression_stmt ::= starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not None, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is None, so that procedure calls do not cause any output.)
7.2. Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

Assignment statements are defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section The standard type hierarchy).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.
• If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.

• Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.
  ○ If the target list contains one target prefixed with an asterisk, called a “starred” target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
  ○ Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

• If the target is an identifier (name):
  ○ If the name does not occur in a global or nonlocal statement in the current code block: the name is bound to the object in the current local namespace.
  ○ Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by nonlocal, respectively.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

• If the target is an attribute reference: The primary expression in
the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the RHS expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The LHS target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the RHS expression refers to a class attribute, the LHS creates a new instance attribute as the target of the assignment:

```python
class Cls:
    x = 3  # class variable
inst = Cls()
inst.x = inst.x + 1  # writes inst.x as 4 leavingCls.x as 3
```

This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is
asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping’s key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

For user-defined objects, the `__setitem__()` method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence’s length. The bounds should evaluate to integers. If either bound is negative, the sequence’s length is added to it. The resulting bounds are clipped to lie between zero and the sequence’s length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

**CPython implementation detail:** In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.
Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are ‘simultaneous’ (for example $a, b = b, a$ swaps two variables), overlaps within the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints $[0, 2]$:

```python
x = [0, 1]
i = 0
i, x[i] = 1, 2  # i is updated, then x[i] is updated
print(x)
```

See also:

**PEP 3132** - Extended Iterable Unpacking

The specification for the *target feature.

### 7.2.1. Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```markdown
augmented_assignment_stmt ::= augtarget augop (expression_list)
augtarget ::= identifier | attribut
augop ::= "+=" | "-=" | "*=" | 
| ">>=" | "<<=" | "&=
```

(See section **Primaries** for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression
list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like \( x += 1 \) can be rewritten as \( x = x + 1 \) to achieve a similar, but not exactly equal effect. In the augmented version, \( x \) is only evaluated once. Also, when possible, the actual operation is performed \textit{in-place}, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side \textit{before} evaluating the right-hand side. For example, \( a[i] += f(x) \) first looks-up \( a[i] \), then it evaluates \( f(x) \) and performs the addition, and lastly, it writes the result back to \( a[i] \).

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible \textit{in-place} behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same caveat about class and instance attributes applies as for regular assignments.

7.2.2. Annotated assignment statements

\textbf{Annotation} assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:
The difference from normal Assignment statements is that only single target and only single right hand side value is allowed.

For simple names as assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute __annotations__ that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

For expressions as assignment targets, the annotations are evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last __setitem__() or __setattr__() call.

See also:

PEP 526 - Syntax for Variable Annotations
The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.

PEP 484 - Type hints
The proposal that added the typing module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.
7.3. The assert statement

Assert statements are a convenient way to insert debugging assertions into a program:

```python
assert_stmt ::= "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```python
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```python
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.
7.4. The pass statement

pass_stmt ::= "pass"

pass is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```python
def f(arg): pass  # a function that does nothing (yet)
class C: pass     # a class with no methods (yet)
```
7.5. The del statement

```
del_stmt ::= "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

*Changed in version 3.2:* Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.
7.6. The return statement

```
return_stmt ::= "return" [expression_list]
```

*return* may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else *None* is substituted.

*return* leaves the current function call with the expression list (or *None*) as return value.

When *return* passes control out of a *try* statement with a *finally* clause, that finally clause is executed before really leaving the function.

In a generator function, the *return* statement indicates that the generator is done and will cause *StopIteration* to be raised. The returned value (if any) is used as an argument to construct *StopIteration* and becomes the *StopIteration.value* attribute.

In an asynchronous generator function, an empty *return* statement indicates that the asynchronous generator is done and will cause *StopAsyncIteration* to be raised. A non-empty *return* statement is a syntax error in an asynchronous generator function.
7.7. The yield statement

\[
\text{yield_stmt ::= yield_expression}
\]

A `yield` statement is semantically equivalent to a `yield expression`. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

\[
yield <\text{expr}>
yield \text{from} <\text{expr}>
\]

are equivalent to the yield expression statements

\[
(yield <\text{expr}>)
(yield \text{from} <\text{expr}>)
\]

Yield expressions and statements are only used when defining a `generator` function, and are only used in the body of the generator function. Using `yield` in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of `yield` semantics, refer to the Yield expressions section.
7.8. The raise statement

```
raise_stmt ::= "raise" [expression ["from" expression]]
```

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The `type` of the exception is the exception instance’s class, the `value` is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute, which is writable. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second `expression` must be another exception class or instance, which will then be attached to the raised exception as the `__cause__` attribute (which is writable). If the raised exception is not handled, both
exceptions will be printed:

```python
>>> try:
...    print(1 / 0)
... except Exception as exc:
...    raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if an exception is raised inside an exception handler or a `finally` clause: the previous exception is then attached as the new exception’s `__context__` attribute:

```python
>>> try:
...    print(1 / 0)
... except:
...    raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```
Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```python
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section `Exceptions`, and information about handling exceptions is in section `The try statement`.

*Changed in version 3.3:* `None` is now permitted as `Y` in `raise X from Y`.

*New in version 3.3:* The `__suppress_context__` attribute to suppress automatic display of the exception context.
7.9. The break statement

\[
\text{break_stmt ::= "break"}
\]

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

When `break` passes control out of a `try` statement with a `finally` clause, that finally clause is executed before really leaving the loop.
7.10. The continue statement

\[
\text{continue_stmt ::= "continue"}
\]

continue may only occur syntactically nested in a for or while loop, but not nested in a function or class definition or finally clause within that loop. It continues with the next cycle of the nearest enclosing loop.

When continue passes control out of a try statement with a finally clause, that finally clause is executed before really starting the next loop cycle.
7.11. The import statement

```
import_stmt ::=
    "import" module ["as" identifier |
    "from" relative_module "import"
        ("," identifier ["as" identifier |
        "from" relative_module "import"
        ("," identifier ["as" identifier |
        "from" module "import" "]"
    module ::= (identifier ".")* identifier
relative_module ::= "."* module | "."+
```

The basic import statement (no from clause) is executed in two steps:

1. find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the import statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules are described in greater detail in the section on the import system, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, or that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:
• If the module name is followed by as, then the name following as is bound directly to the imported module.
• If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module.
• If the module being imported is not a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly.

The from form uses a slightly more complex process:

1. find the module specified in the from clause, loading and initializing it if necessary;
2. for each of the identifiers specified in the import clauses:
   1. check if the imported module has an attribute by that name
   2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute
   3. if the attribute is not found, ImportError is raised.
   4. otherwise, a reference to that value is stored in the local namespace, using the name in the as clause if it is present, otherwise using the attribute name.

Examples:

```python
import foo  # foo imported and bound
import foo.bar.baz  # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz  # foo.bar.baz imported and baz bound
from foo import attr  # foo imported and foo.attr bound
```

If the list of identifiers is replaced by a star ('*'), all public names defined in the module are bound in the local namespace for the scope
where the `import` statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — `from module import *` — is only allowed at the module level. Attempting to use it in class or function definitions will raise a `SyntaxError`.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after `from` you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained within [PEP 328](https://www.python.org/dev/peps/pep-0328/).
importlib.import_module() is provided to support applications that determine dynamically the modules to be loaded.

7.11.1. Future statements

A future statement is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_stmt ::= "from" "__future__" "import" feature ("," feature ["as" identifier])*  
| "from" "__future__" "import" "(" 
| ("," feature ["as" identifier])* [""

feature ::= identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The only feature in Python 3.7 that requires using the future statement is annotations.

All historical features enabled by the future statement are still
recognized by Python 3. The list includes `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it’s an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and
compile() that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to compile() — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the -i option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

See also:

PEP 236 - Back to the __future__
  The original proposal for the __future__ mechanism.
7.12. The global statement

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared `global`.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, `class` definition, function definition, `import` statement, or variable annotation.

**CPython implementation detail:** The current implementation does not enforce some of these restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

**Programmer’s note:** `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block containing the function call, and code contained in such a string is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.
7.13. The nonlocal statement

nonlocal_stmt ::= "nonlocal" identifier ("," identifier)*

The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. This is important because the default behavior for binding is to search the local namespace first. The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope.

Names listed in a `nonlocal` statement, unlike those listed in a `global` statement, must refer to pre-existing bindings in an enclosing scope (the scope in which a new binding should be created cannot be determined unambiguously).

Names listed in a `nonlocal` statement must not collide with pre-existing bindings in the local scope.

See also:

**PEP 3104 - Access to Names in Outer Scopes**

The specification for the `nonlocal` statement.
8. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The \texttt{if}, \texttt{while} and \texttt{for} statements implement traditional control flow constructs. \texttt{try} specifies exception handlers and/or cleanup code for a group of statements, while the \texttt{with} statement allows the execution of initialization and finalization code around a block of code. Function and class definitions are also syntactically compound statements.

A compound statement consists of one or more ‘clauses.’ A clause consists of a header and a ‘suite.’ The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header’s colon, or it can be one or more indented statements on subsequent lines. Only the latter form of a suite can contain nested compound statements; the following is illegal, mostly because it wouldn’t be clear to which \texttt{if} clause a following \texttt{else} clause would belong:

\begin{verbatim}
if test1: if test2: print(x)
\end{verbatim}

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the \texttt{print()} calls are executed:
if x < y < z: print(x); print(y); print(z)

Summarizing:

\[
\text{compound_stmt} ::= \text{if_stmt} \\
\hspace{1em} \mid \text{while_stmt} \\
\hspace{1em} \mid \text{for_stmt} \\
\hspace{1em} \mid \text{try_stmt} \\
\hspace{1em} \mid \text{with_stmt} \\
\hspace{1em} \mid \text{funcdef} \\
\hspace{1em} \mid \text{classdef} \\
\hspace{1em} \mid \text{async_with_stmt} \\
\hspace{1em} \mid \text{async_for_stmt} \\
\hspace{1em} \mid \text{async_funcdef}
\]

\[
\text{suite} ::= \text{stmt_list} \text{ NEWLINE} | \text{NEWLINE INDENT}
\]

\[
\text{statement} ::= \text{stmt_list} \text{ NEWLINE} | \text{compound_stmt}
\]

\[
\text{stmt_list} ::= \text{simple_stmt} (";" \text{ simple_stmt})\star [\text{ NEWLINE}]
\]

Note that statements always end in a \text{NEWLINE} possibly followed by a \text{DEDENT}. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the ‘dangling else’ problem is solved in Python by requiring nested \text{if} statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.
8.1. The if statement

The if statement is used for conditional execution:

```
if_stmt ::= "if" expression ":" suite
           ("elif" expression ":" suite)*
           ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the if statement is executed or evaluated). If all expressions are false, the suite of the else clause, if present, is executed.
8.2. The while statement

The `while` statement is used for repeated execution as long as an expression is true:

```
while_stmt ::= "while" expression ":" suite
["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the else clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.
8.3. The for statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::= "for" target_list "in" expression_list
            ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the `else` clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there is no next item.

The for-loop makes assignments to the variables(s) in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
i = 5                    # this will not affect the for
# because i will be overwritten
# index in the range
```
Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns an iterator of integers suitable to emulate the effect of Pascal’s `for i := a to b do;` e.g., `list(range(3))` returns the list `[0, 1, 2]`.

**Note:** There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```python
for x in a[:]:
    if x < 0: a.remove(x)
```
8.4. The try statement

The **try** statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" suite
           | "try"
           | ("except" [expression ["as" identifier]
                     ["else" ":" suite]
                     ["finally" ":" suite])
try2_stmt ::= "try" ":" suite
            | "try"
            | ("finally" ":" suite)
```

The **except** clause(s) specify one or more exception handlers. When no exception occurs in the **try** clause, no exception handler is executed. When an exception occurs in the **try** suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is the class or a base class of the exception object or a tuple containing an item compatible with the exception.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. [1]

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the
call stack (it is treated as if the entire `try` statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified after the `as` keyword in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire `try` statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the `try` clause of the inner handler, the outer handler will not handle the exception.)

When an exception has been assigned using `as target`, it is cleared at the end of the except clause. This is as if

```python
except E as N:
    foo
```

was translated to

```python
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an except clause’s suite is executed, details about the exception are stored in the `sys` module and can be accessed via
sys.exc_info(). sys.exc_info() returns a 3-tuple consisting of the exception class, the exception instance and a traceback object (see section The standard type hierarchy) identifying the point in the program where the exception occurred. sys.exc_info() values are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional else clause is executed if the control flow leaves the try suite, no exception was raised, and no return, continue, or break statement was executed. Exceptions in the else clause are not handled by the preceding except clauses.

If finally is present, it specifies a ‘cleanup’ handler. The try clause is executed, including any except and else clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The finally clause is executed. If there is a saved exception it is re-raised at the end of the finally clause. If the finally clause raises another exception, the saved exception is set as the context of the new exception. If the finally clause executes a return or break statement, the saved exception is discarded:

```python
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
... f()
42
```

The exception information is not available to the program during execution of the finally clause.
When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try...finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

```
>>> def foo():
    ...    try:
    ...        return 'try'
    ...    finally:
    ...        return 'finally'
    ...
>>> foo()
'finally'
```

Additional information on exceptions can be found in section `Exceptions`, and information on using the `raise` statement to generate exceptions may be found in section `The raise statement`. 
8.5. The with statement

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common `try...except...finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt ::= "with" with_item ("," with_item)* ":"
with_item ::= expression ["as" target]
```

The execution of the `with` statement with one “item” proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.

2. The context manager’s `__exit__()` is loaded for later use.

3. The context manager’s `__enter__()` method is invoked.

4. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

   Note: The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.

5. The suite is executed.
6. The context manager's `__exit__() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()` . Otherwise, three `None` arguments are supplied.

If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

With more than one item, the context managers are processed as if multiple `with` statements were nested:

```python
with A() as a, B() as b:
    suite
```

is equivalent to

```python
with A() as a:
    with B() as b:
        suite
```

*Changed in version 3.1:* Support for multiple context expressions.

*See also:*

**PEP 343** - The “with” statement

The specification, background, and examples for the Python
with statement.
### 8.6. Function definitions

A function definition defines a user-defined function object (see section The standard type hierarchy):

| funcdef | ::= | [decorators] "def" funcname ["->" expression] ":" suite |
| decorators | ::= | decorator+ |
| decorator | ::= | "@" dotted_name ["(" [argument_list dotted_name] ")" ["(" parameter_list ")"] |
| dotted_name | ::= | identifier ("." identifier) |
| parameter_list | ::= | defparameter ("," defparameter | parameter_list_starargs |
| parameter_list_starargs | ::= | "*" [parameter] ("," defparameter | "**" parameter ["]") |
| parameter | ::= | identifier [":" expression] |
| defparameter | ::= | parameter ["=" expression] |
| funcname | ::= | identifier |

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called. [2]

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested...
fashion. For example, the following code

```python
@f1(arg)
@f2
def func(): pass
```

is roughly equivalent to

```python
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

When one or more parameters have the form `parameter = expression`, the function is said to have “default parameter values.” For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the “*” must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

**Default parameter values are evaluated from left to right when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```python
def whats_on_the_telly(penguin=None):
```
if penguin is None:
    penguin = []
penguin.append("property of the zoo")
return penguin

Function call semantics are described in more detail in section Calls. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form “*identifier” is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form “**identifier” is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after “*” or “*identifier” are keyword-only parameters and may only be passed used keyword arguments.

Parameters may have an annotation of the form “: expression” following the parameter name. Any parameter may have an annotation, even those of the form *identifier or **identifier. Functions may have “return” annotation of the form “-> expression” after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters’ names in the __annotations__ attribute of the function object. If the annotations import from __future__ is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.
It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section **Lambdas**. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a “def” statement can be passed around or assigned to another name just like a function defined by a lambda expression. The “def” form is actually more powerful since it allows the execution of multiple statements and annotations.

**Programmer’s note:** Functions are first-class objects. A “def” statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section **Naming and binding** for details.

### See also:

**PEP 3107 - Function Annotations**  
The original specification for function annotations.

**PEP 484 - Type Hints**  
Definition of a standard meaning for annotations: type hints.

**PEP 526 - Syntax for Variable Annotations**  
Ability to type hint variable declarations, including class variables and instance variables

**PEP 563 - Postponed Evaluation of Annotations**  
Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.
8.7. Class definitions

A class definition defines a class object (see section The standard type hierarchy):

\[
\text{classdef ::= [decorators] "class" classname [inheritance]}
\]

\[
\text{inheritance ::= "(" [argument_list] ")"}
\]

\[
\text{classname ::= identifier}
\]

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see Metaclasses for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class object; hence,

```
class Foo:
    pass
```

is equivalent to

```
class Foo(object):
    pass
```

The class’s suite is then executed in a new execution frame (see Naming and binding), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class’s suite finishes execution, its execution frame is discarded but its local namespace is saved. [3] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.
The order in which attributes are defined in the class body is preserved in the new class's `__dict__`. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using `metaclasses`.

Classes can also be decorated: just like when decorating functions,

```python
@f1(arg)
@f2
class Foo: pass
```

is roughly equivalent to

```python
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

**Programmer’s note:** Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation “`self.name`”, and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results.Descriptors can be used to create instance variables with different implementation details.

**See also:**

**PEP 3115 - Metaclasses in Python 3000**
The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

**PEP 3129 - Class Decorators**

The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.
8.8. Coroutines

New in version 3.5.

8.8.1. Coroutine function definition

```
async_funcdef ::= [decorators] "async" "def" funcname
                ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see coroutine). Inside the body of a coroutine function, await and async identifiers become reserved keywords; await expressions, async for and async with can only be used in coroutine function bodies.

Functions defined with async def syntax are always coroutine functions, even if they do not contain await or async keywords.

It is a SyntaxError to use a yield from expression inside the body of a coroutine function.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2. The async for statement

```
async_for_stmt ::= "async" for_stmt
```
An **asynchronous iterable** is able to call asynchronous code in its *iter* implementation, and **asynchronous iterator** can call asynchronous code in its *next* method.

The **async for** statement allows convenient iteration over asynchronous iterators.

The following code:

```python
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

Is semantically equivalent to:

```python
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True
while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        BLOCK
else:
    BLOCK2
```

See also **__aiter__()** and **__anext__()** for details.

It is a **SyntaxError** to use an async for statement outside the body of a coroutine function.

**8.8.3. The async with statement**
An asynchronous context manager is a context manager that is able to suspend execution in its enter and exit methods.

The following code:

```python
async with EXPR as VAR:
    BLOCK
```

Is semantically equivalent to:

```python
mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)
VAR = await aenter
try:
    BLOCK
except:
    if not await aexit(mgr, *sys.exc_info()):
        raise
else:
    await aexit(mgr, None, None, None)
```

See also `__aenter__()` and `__aexit__()` for details.

It is a SyntaxError to use an async with statement outside the body of a coroutine function.

See also:

**PEP 492 - Coroutines with async and await syntax**
The proposal that made coroutines a proper standalone concept
in Python, and added supporting syntax.

Footnotes

The exception is propagated to the invocation stack unless there is a `finally` clause which happens to raise another exception. That new exception causes the old one to be lost.

A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's docstring.

A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's docstring.
9. Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.
9.1. Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `builtins` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

A complete program can be passed to the interpreter in three forms: with the `-c string` command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.
9.2. File input

All input read from non-interactive files has the same form:

```
file_input ::= (NEWLINE | statement)*
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec()` function;
9.3. Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input ::= [stmt_list] NEWLINE | compound_stmt
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.
9.4. Expression input

`eval()` is used for expression input. It ignores leading whitespace. The string argument to `eval()` must have the following form:

```
expr_input ::= expression_list NEWLINE*
```

10. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```plaintext
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
# single_input is a single interactive statement
# file_input is a module or sequence of commands read from an input file
# eval_input is the input for the eval() function
# NB: compound_stmt in single_input is followed by a
single_input: NEWLINE | simple_stmt | compound_stmt
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: 'async' funcdef
funcdef: 'def' NAME parameters [ '->' test ] ':' suite

parameters: '(' [typedargslist] ')
typedargslist: (tfpdef [ '=' test ] (',' tfpdef [ '=' test ])* [',', ['|' '*'] tfpdef [',']])
| '* ' tfpdef [','])
| '** ' tfpdef [','])
tfpdef: NAME [ ':' test ]
varargslist: (vfpdef [ '=' test ] (',' vfpdef [ '=' test ])* [',', ['|' '*'] vfpdef [',']])
```
```plaintext
vfpdef: NAME

stmt: simple_stmt | compound_stmt

simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE

small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt | import_stmt | global_stmt | nonlocal_stmt)

expr_stmt: testlist_star_expr (annassign | augassign)

annassign: ':' test ['=' test]

testlist_star_expr: (test|star_expr) (',' (test|star_expr))

augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '<<=' | '>>=' | '***=' | '/=')

# For normal and annotated assignments, additional restrictions
# For normal and annotated assignments, additional restrictions

del_stmt: 'del' exprlist

pass_stmt: 'pass'

flow_stmt: break_stmt | continue_stmt | return_stmt

break_stmt: 'break'

continue_stmt: 'continue'

return_stmt: 'return' [testlist]

yield_stmt: yield_expr

raise_stmt: 'raise' [test ['from' test]]

import_stmt: import_name | import_from

import_name: 'import' dotted_as_names

# note below: the ('.' | '...') is necessary because
import_from: ('from' (('.') | '...')*) dotted_name |

import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)
dotted_as_names: dotted_as_name (',' dotted_as_name)
dotted_name: NAME ('.') NAME)*

global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
```

assert_stmt: 'assert' test [','] test

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | async_stmt
async_stmt: 'async' (funcdef | with_stmt | for_stmt)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)*
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
   (except_clause ':' suite)+
   ['else' ':' suite]
   ['finally' ':' suite] |
   'finally' ':' suite))
with_stmt: 'with' with_item (',', with_item)* ':' suite
with_item: test [as' expr]

# NB compile.c makes sure that the default except clause
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# <> isn't actually a valid comparison operator in Python
# sake of a __future__ import described in PEP 401 (comp_op: '<' | '<=' | '==' | '<' | '<>' | '!=' | 'in' | 'not'
star_expr: '*' expr
eq: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr ('<<' | '>>') arith_expr*
arith_expr: term ('+' | '-' | '*' | '@' | '/' | '%' | '//' | '**') term*
term: factor ('*' | '@' | '/' | '%' | '//' | '**') factor*
factor: ('+' | '-' | '~') factor | power
power: atom_expr [ '**' factor]
atom_expr: ['await'] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')') |
      ['testlist_comp'] |
      '{' [dictorsetmaker] '}' |
      NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False' | 'None'
testlist_comp: (test|star_expr) ( comp_for | (',', (ttrailer: ('(' [arglist] ')') | ['' subscriptlist '']) |
subscriptlist: subscript (',', 'subscript)* ['',']
subscript: test | [test] '::' [test] [sliceop]
sliceop: '::' [test]
exprlist: (expr|star_expr) (',', (expr|star_expr))* ['',']
testlist: test (',', test)* ['',']
dictorsetmaker: ( ((test '::' test | '**' expr)
     (comp_for | (',', (test '::' test |
     ((test | star_expr)
     (comp_for | (',', (test | star_expr)
classdef: 'class' NAME ['(' [arglist] ')'] '::' suite
arglist: argument (',', argument)* ['','']

# The reason that keywords are test nodes instead of NAME results in an ambiguity. ast.c makes sure it's a NAME.
# "test '==' test" is really "keyword '==' test", but # These need to be in a single rule to avoid grammar # to our LL(1) parser. Even though 'test' includes 'test'
# we explicitly match '*' here, too, to give it proper # Illegal combinations and orderings are blocked in # multiple (test comp_for) arguments are blocked; keyword that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
      test '==' test |
      '**' test |
      '* ' test )
comp_iter: comp_for | comp_if
sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_for: ['async'] sync_comp_for
comp_if: 'if' test_nocond [comp_iter]

# not used in grammar, but may appear in "node" pass
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
Introduction

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an import statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the
built-in functions, data types and exceptions, and finally the modules, grouped in chapters of related modules.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don't have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module \texttt{random}) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter \textbf{Built-in Functions}, as the remainder of the manual assumes familiarity with this material.

Let the show begin!
Notes on availability

- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.
Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>delattr()</td>
<td>hash()</td>
<td>memoryview</td>
</tr>
<tr>
<td>all()</td>
<td>dict()</td>
<td>help()</td>
<td>min()</td>
</tr>
<tr>
<td>any()</td>
<td>dir()</td>
<td>hex()</td>
<td>next()</td>
</tr>
<tr>
<td>ascii()</td>
<td>divmod()</td>
<td>id()</td>
<td>object()</td>
</tr>
<tr>
<td>bin()</td>
<td>enumerate()</td>
<td>input()</td>
<td>oct()</td>
</tr>
<tr>
<td>bool()</td>
<td>eval()</td>
<td>int()</td>
<td>open()</td>
</tr>
<tr>
<td>breakpoint()</td>
<td>exec()</td>
<td>instanceof()</td>
<td>ord()</td>
</tr>
<tr>
<td>bytearray()</td>
<td>filter()</td>
<td>issubclass()</td>
<td>pow()</td>
</tr>
<tr>
<td>bytes()</td>
<td>float()</td>
<td>iter()</td>
<td>print()</td>
</tr>
<tr>
<td>callable()</td>
<td>format()</td>
<td>len()</td>
<td>property()</td>
</tr>
<tr>
<td>chr()</td>
<td>frozenset()</td>
<td>list()</td>
<td>range()</td>
</tr>
<tr>
<td>classmethod()</td>
<td>getattr()</td>
<td>locals()</td>
<td>repr()</td>
</tr>
<tr>
<td>compile()</td>
<td>globals()</td>
<td>map()</td>
<td>reversed()</td>
</tr>
<tr>
<td>complex()</td>
<td>hasattr()</td>
<td>max()</td>
<td>round()</td>
</tr>
</tbody>
</table>

**abs(x)**

Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.
all(iterable)

Return True if all elements of the iterable are true (or if the iterable is empty). Equivalent to:

```python
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any(iterable)

Return True if any element of the iterable is true. If the iterable is empty, return False. Equivalent to:

```python
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(object)

As repr(), return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by repr() using \x, \u or \U escapes. This generates a string similar to that returned by repr() in Python 2.

bin(x)

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If x is not a Python int object, it has to define an __index__() method that returns an integer. Some examples:

```python
>>> bin(3)
```
If prefix “0b” is desired or not, you can use either of the following ways.

```python
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also `format()` for more information.

```python
class bool([x])
```

Return a Boolean value, i.e. one of `True` or `False`. `x` is converted using the standard truth testing procedure. If `x` is false or omitted, this returns `False`; otherwise it returns `True`. The `bool` class is a subclass of `int` (see Numeric Types — `int`, `float`, `complex`). It cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Values).

*Changed in version 3.7:* `x` is now a positional-only parameter.

```python
breakpoint(*args, **kws)
```

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don’t have to explicitly import `pdb` or `type` as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and
Breakpoint() will automatically call that, allowing you to drop into the debugger of choice.

New in version 3.7.

Class `bytearray([source[, encoding[, errors]]])`

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in `Mutable Sequence Types`, as well as most methods that the `bytes` type has, see `Bytes and Bytearray Operations`.

The optional `source` parameter can be used to initialize the array in a few different ways:

- If it is a string, you must also give the `encoding` (and optionally, `errors`) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an integer, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the `buffer` interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an iterable, it must be an iterable of integers in the range 0 <= x < 256, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also Binary Sequence Types — `bytes`, `bytearray`, `memoryview` and `Bytearray Objects`.

Class `bytes([source[, encoding[, errors]]])`

Return a new “bytes” object, which is an immutable sequence of integers in the range 0 <= x < 256. `bytes` is an immutable
version of bytearray – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for bytearray().

Bytes objects can also be created with literals, see String and Bytes literals.

See also Binary Sequence Types — bytes, bytearray, memoryview, Bytes Objects, and Bytes and Bytearray Operations.

callable(object)
Return True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a __call__() method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr(i)
Return the string representing a character whose Unicode code point is the integer i. For example, chr(97) returns the string 'a', while chr(8364) returns the string '€'. This is the inverse of ord().

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). ValueError will be raised if i is outside that range.

@classmethod
Transform a method into a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```python
class C:
    @classmethod
def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function decorator – see the description of function definitions in Function definitions for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()` in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in The standard type hierarchy.

`compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)`

Compile the `source` into a code or AST object. Code objects can be executed by `exec()` or `eval()`. `source` can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn’t read from a file.
('string>' is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be 'exec' if *source* consists of a sequence of statements, 'eval' if it consists of a single expression, or 'single' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than None will be printed).

The optional arguments *flags* and *dont_inherit* control which future statements affect the compilation of *source*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the `compiler_flag` attribute on the _Feature instance in the __future__ module.

The argument *optimize* specifies the optimization level of the compiler; the default value of -1 selects the optimization level of the interpreter as given by -O options. Explicit levels are 0 (no optimization; **__debug__** is true), 1 (asserts are removed, **__debug__** is false) or 2 (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.
If you want to parse Python code into its AST representation, see `ast.parse()`.

**Note:** When compiling a string with multi-line code in 'single' or 'eval' mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the code module.

**Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python’s AST compiler.

*Changed in version 3.2:* Allowed use of Windows and Mac newlines. Also input in 'exec' mode does not have to end in a newline anymore. Added the `optimize` parameter.

*Changed in version 3.5:* Previously, `TypeError` was raised when null bytes were encountered in source.

```
class complex([real[, imag]])
```

Return a complex number with the value `real + imag*1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If `imag` is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

**Note:** When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises
ValueError.

The complex type is described in Numeric Types — int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

delattr(object, name)
This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar.

class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
Create a new dictionary. The dict object is the dictionary class. See dict and Mapping Types — dict for documentation about this class.

For other containers see the built-in list, set, and tuple classes, as well as the collections module.

dir([object])
Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named __dir__(), this method will be called and must return the list of attributes. This allows objects that implement a custom __getattr__() or __getattribute__()
function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object’s `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`. The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module’s attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object’s attributes’ names, the names of its class’s attributes, and recursively of the attributes of its class’s base classes.

The resulting list is sorted alphabetically. For example:

```python
>>> import struct
>>> dir()  # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)  # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__initializing__', '__loader__', '__name__', '__clearcache', 'calcsize', 'error', 'pack', 'pack_into', 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
```
Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

```
>>> dir(s)
['area', 'location', 'perimeter']
```

**divmod**(\(a, b\))

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as \((a \div b, a \% b)\). For floating point numbers the result is \((q, a \% b)\), where \(q\) is usually `math.floor(a / b)` but may be 1 less than that. In any case \(q \times b + a \% b\) is very close to \(a\), if \(a \% b\) is non-zero it has the same sign as \(b\), and \(0 \leq \text{abs}(a \% b) < \text{abs}(b)\).

**enumerate**(iterable, start=0)

Return an enumerate object. `iterable` must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from `start` which defaults to 0) and the values obtained from iterating over `iterable`.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

Equivalent to:

def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
    n += 1

eval(expression, globals=None, locals=None)

The arguments are a string and optional globals and locals. If provided, `globals` must be a dictionary. If provided, `locals` can be any mapping object.

The `expression` argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the `globals` and `locals` dictionaries as global and local namespace. If the `globals` dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before `expression` is parsed. This means that `expression` normally has full access to the standard `builtins` module and restricted environments are propagated. If the `locals` dictionary is omitted it defaults to the `globals` dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> eval('x+1')
2
```
This function can also be used to execute arbitrary code objects (such as those created by \texttt{compile()}). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the mode argument, \texttt{eval()}'s return value will be None.

Hints: dynamic execution of statements is supported by the \texttt{exec()} function. The \texttt{globals()} and \texttt{locals()} functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by \texttt{eval()} or \texttt{exec()}

See \texttt{ast.literal_eval()} for a function that can safely evaluate strings with expressions containing only literals.

\texttt{exec(object[, globals[, locals]])}

This function supports dynamic execution of Python code. \texttt{object} must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section “File input” in the Reference Manual). Be aware that the \texttt{return} and \texttt{yield} statements may not be used outside of function definitions even within the context of code passed to the \texttt{exec()} function. The return value is \texttt{None}.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only \texttt{globals} is provided, it must be a dictionary, which will be used for both the global and the local variables. If \texttt{globals} and \texttt{locals} are given, they are used for the global and local variables, respectively. If provided, \texttt{locals} can be any mapping object. Remember that at module level, globals and
locals are the same dictionary. If exec gets two separate objects as `globals` and `locals`, the code will be executed as if it were embedded in a class definition.

If the `globals` dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into `globals` before passing it to `exec()`.

**Note:** The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

**Note:** The default `locals` act as described for function `locals()` below: modifications to the default `locals` dictionary should not be attempted. Pass an explicit `locals` dictionary if you need to see effects of the code on `locals` after function `exec()` returns.

`filter(function, iterable)`

Construct an iterator from those elements of `iterable` for which `function` returns true. `iterable` may be either a sequence, a container which supports iteration, or an iterator. If `function` is `None`, the identity function is assumed, that is, all elements of `iterable` that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))`. 
function(item)) if function is not None and (item for item in iterable if item) if function is None.

See `itertools.filterfalse()` for the complementary function that returns elements of `iterable` for which `function` returns false.

class `float`([`x`])

Return a floating point number constructed from a number or string `x`.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign ::= "+" | "-
infinity ::= "Infinity" | "inf"
nan ::= "nan"
numeric_value ::= floatnumber | infinity | nan
numeric_string ::= [sign] numeric_value
```

Here `floatnumber` is the form of a Python floating-point literal, described in Floating point literals. Case is not significant, so, for example, “inf”, “Inf”, “INFINITY” and “iNfINity” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python's
floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If no argument is given, `0.0` is returned.

Examples:

```python
>>> float('+1.23')
1.23
>>> float(' -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

The float type is described in `Numeric Types — int, float, complex`

`Changed in version 3.6:` Grouping digits with underscores as in code literals is allowed.

`Changed in version 3.7:` `x` is now a positional-only parameter.

`format(value[, format_spec])`
Convert a `value` to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the `value` argument, however there is a standard formatting syntax that is used by most built-in types: `Format Specification Mini-Language`.
The default *format_spec* is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value's `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the `format_spec` is non-empty, or if either the `format_spec` or the return value are not strings.

*Changed in version 3.4:* `object().__format__(format_spec)` raises `TypeError` if `format_spec` is not an empty string.

class `frozenset`([`iterable`])

Return a new `frozenset` object, optionally with elements taken from `iterable`. `frozenset` is a built-in class. See `frozenset` and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`getattr(object, name[, default])`

Return the value of the named attribute of `object`. `name` must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

`globals()`
Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

**hasattr**(object, name)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.)

**hash**(object)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

**Note:** For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine. See `__hash__()` for details.

**help**([object])

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(`/`) appears in the parameter list of a function,
when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the `site` module.

*Changed in version 3.4:* Changes to `pydoc` and `inspect` mean that the reported signatures for callables are now more comprehensive and consistent.

**hex(x)**

Convert an integer number to a lowercase hexadecimal string prefixed with “0x”. If x is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```python
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lowercase hexadecimal string with prefix or not, you can use either of the following ways:

```python
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

See also `format()` for more information.
See also `int()` for converting a hexadecimal string to an integer using a base of 16.

**Note:** To obtain a hexadecimal string representation for a float, use the `float.hex()` method.

**id**(object)
Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

**CPython implementation detail:** This is the address of the object in memory.

**input**([prompt])
If the `prompt` argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```python
>>> s = input('--> ')  
---> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

```python
class int([x])
class int(x, base=10)
```

Return an integer object constructed from a number or string `x`, or
return 0 if no arguments are given. If \( x \) defines \(_\text{int}\)( ), \( \text{int}(x) \) returns \( x._\text{int}() \). If \( x \) defines \(_\text{trunc}\)( ), it returns \( x._\text{trunc}() \). For floating point numbers, this truncates towards zero.

If \( x \) is not a number or if \( \text{base} \) is given, then \( x \) must be a string, \text{bytes}, or \text{bytearray} instance representing an integer literal in radix \( \text{base} \). Optionally, the literal can be preceded by + or - (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having values 10 to 35. The default \( \text{base} \) is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with 0b/0B, 0o/0O, or 0x/0X, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that \( \text{int}('010', 0) \) is not legal, while \( \text{int}('010') \) is, as well as \( \text{int}('010', 8) \).

The integer type is described in \text{Numeric Types — int, float, complex}.

\textit{Changed in version 3.4:} If \( \text{base} \) is not an instance of \text{int} and the \( \text{base} \) object has a \text{base._index__} method, that method is called to obtain an integer for the base. Previous versions used \text{base._int__} instead of \text{base._index__}.

\textit{Changed in version 3.6:} Grouping digits with underscores as in code literals is allowed.

\textit{Changed in version 3.7:} \( x \) is now a positional-only parameter.

\texttt{isinstance}(\texttt{object, classinfo})

Return true if the \texttt{object} argument is an instance of the \texttt{classinfo}
argument, or of a (direct, indirect or virtual) subclass thereof. If object is not an object of the given type, the function always returns false. If classinfo is a tuple of type objects (or recursively, other such tuples), return true if object is an instance of any of the types. If classinfo is not a type or tuple of types and such tuples, a TypeError exception is raised.

issubclass(class, classinfo)
Return true if class is a subclass (direct, indirect or virtual) of classinfo. A class is considered a subclass of itself. classinfo may be a tuple of class objects, in which case every entry in classinfo will be checked. In any other case, a TypeError exception is raised.

iter(object[, sentinel])
Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, object must be a collection object which supports the iteration protocol (the __iter__() method), or it must support the sequence protocol (the __getitem__() method with integer arguments starting at 0). If it does not support either of those protocols, TypeError is raised. If the second argument, sentinel, is given, then object must be a callable object. The iterator created in this case will call object with no arguments for each call to its __next__() method; if the value returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.

See also Iterator Types.

One useful application of the second form of iter() is to build a block-reader. For example, reading fixed-width blocks from a binary
database file until the end of file is reached:

```python
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):  
        process_block(block)
```

**len(s)**

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

**class list(iterable)**

Rather than being a function, `list` is actually a mutable sequence type, as documented in Lists and Sequence Types — list, tuple, range.

**locals()**

Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks.

**Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

**map(function, iterable, ...)**

Return an iterator that applies `function` to every item of `iterable`, yielding the results. If additional `iterable` arguments are passed, `function` must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the
function inputs are already arranged into argument tuples, see `itertools.starmap()`.

```
max(iterable, *, [key, default])
max(arg1, arg2, *args[, key])
```

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The `key` argument specifies a one-argument ordering function like that used for `list.sort()`. The `default` argument specifies an object to return if the provided iterable is empty. If the iterable is empty and `default` is not provided, a `ValueError` is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc).

*New in version 3.4:* The `default` keyword-only argument.

```
memoryview(obj)
```

Return a “memory view” object created from the given argument. See Memory Views for more information.

```
min(iterable, *, [key, default])
```
**min**(*arg1, arg2, *args[, key]*)

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The key argument specifies a one-argument ordering function like that used for list.sort(). The default argument specifies an object to return if the provided iterable is empty. If the iterable is empty and default is not provided, a ValueError is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as sorted(iterable, key=keyfunc)[0] and heapq.nsmallest(1, iterable, key=keyfunc).

*New in version 3.4:* The default keyword-only argument.

**next**(*iterator[, default]*)

Retrieve the next item from the iterator by calling its __next__() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised.

**class object**

Return a new featureless object. object is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.
Note: `object` does not have a `__dict__`, so you can’t assign arbitrary attributes to an instance of the `object` class.

**oct**(*x*)

Convert an integer number to an octal string prefixed with “0o”. The result is a valid Python expression. If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:

```python
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to octal string either with prefix “0o” or not, you can use either of the following ways.

```python
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

See also `format()` for more information.

**open**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Open *file* and return a corresponding file object. If the file cannot be opened, an `OSError` is raised.

*file* is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer
file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for exclusive creation and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>open for reading (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>open for writing, truncating the file first</td>
</tr>
<tr>
<td>'x'</td>
<td>open for exclusive creation, failing if the file already exists</td>
</tr>
<tr>
<td>'a'</td>
<td>open for writing, appending to the end of the file if it exists</td>
</tr>
<tr>
<td>'b'</td>
<td>binary mode</td>
</tr>
<tr>
<td>'t'</td>
<td>text mode (default)</td>
</tr>
<tr>
<td>'+'</td>
<td>open a disk file for updating (reading and writing)</td>
</tr>
</tbody>
</table>

The default mode is 'r' (open for reading text, synonym of 'rt'). For binary read-write access, the mode 'w+b' opens and
truncates the file to 0 bytes. 'r+b' opens the file without truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including 'b' in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is included in the mode argument), the contents of the file are returned as str, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

There is an additional mode character permitted, 'U', which no longer has any effect, and is considered deprecated. It previously enabled universal newlines in text mode, which became the default behaviour in Python 3.0. Refer to the documentation of the newline parameter for further details.

**Note:** Python doesn’t depend on the underlying operating system’s notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
• “Interactive” text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files.

`encoding` is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any text encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

`errors` is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers), though any error handling name that has been registered with `codecs.register_error()` is also valid. The standard names include:

- `'strict'` to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.
- `'replace'` causes a replacement marker (such as `'?')` to be inserted where there is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`. 
- `'\backslashreplace' replaces malformed data by Python's backslashed escape sequences.
- `'\name\replace' (also only supported when writing) replaces unsupported characters with \N{...} escape sequences.

`newline` controls how universal newlines mode works (it only applies to text mode). It can be None, '', '\n', '\r', and '\r\\n'. It works as follows:

- When reading input from the stream, if `newline` is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if `newline` is None, any '\n' characters written are translated to the system default line separator, `os.lninesep`. If `newline` is '' or '\n', no translation takes place. If `newline` is any of the other legal values, any '\n' characters written are translated to the given string.

If `closefd` is False and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` must be True (the default) otherwise an error will be raised.

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with (file, flags). `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing None).
The newly created file is **non-inheritable**.

The following example uses the `dir_fd` parameter of the `os.open()` function to open a file relative to a given directory:

```python
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spampam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spampam.txt')
>>> os.close(dir_fd)  # don't leak a file descriptor
```

The type of **file object** returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a subclass of `io.TextIOBase` (specifically `io.TextIOWrapper`). When used to open a file in a binary mode with buffering, the returned class is a subclass of `io.BufferedIOBase`. The exact class varies: in read binary mode, it returns an `io.BufferedReader`; in write binary and append binary modes, it returns an `io.BufferedWriter`, and in read/write mode, it returns an `io.BufferedRandom`. When buffering is disabled, the raw stream, a subclass of `io.RawIOBase`, `io.FileIO`, is returned.

See also the file handling modules, such as, `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

*Changed in version 3.3:*
- The `opener` parameter was added.
- The 'x' mode was added.
- IOError used to be raised, it is now an alias of OSError.
- FileExistsError is now raised if the file opened in exclusive creation mode ('x') already exists.

*Changed in version 3.4:*
- The file is now non-inheritable.

*Deprecated since version 3.4, will be removed in version 4.0:* The 'U' mode.

*Changed in version 3.5:*
- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an InterruptedError exception (see PEP 475 for the rationale).
- The 'namereplace' error handler was added.

*Changed in version 3.6:*
- Support added to accept objects implementing os.PathLike.
- On Windows, opening a console buffer may return a subclass of io.RawIOBase other than io.FileI0.

**ord(c)**
Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, ord('a') returns the integer 97 and ord('€') (Euro sign) returns 8364. This is the inverse of chr().

**pow(x, y[, z])**
Return $x$ to the power $y$; if $z$ is present, return $x$ to the power $y$,
modulo \( z \) (computed more efficiently than \( \text{pow}(x, y) \mod z \)). The two-argument form \( \text{pow}(x, y) \) is equivalent to using the power operator: \( x^{**}y \).

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For \textit{int} operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, \( 10^{**}2 \) returns 100, but \( 10^{**}-2 \) returns 0.01. If the second argument is negative, the third argument must be omitted. If \( z \) is present, \( x \) and \( y \) must be of integer types, and \( y \) must be non-negative.

\begin{code}
\texttt{print(*objects, sep='\ ', end='\n', file=sys.stdout, flush=False)}
\end{code}

Print \textit{objects} to the text stream \textit{file}, separated by \textit{sep} and followed by \textit{end}. \textit{sep, end, file} and \textit{flush}, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like \texttt{str()} does and written to the stream, separated by \textit{sep} and followed by \textit{end}. Both \textit{sep} and \textit{end} must be strings; they can also be \texttt{None}, which means to use the default values. If no \textit{objects} are given, \texttt{print()} will just write \textit{end}.

The \textit{file} argument must be an object with a \texttt{write(string)} method; if it is not present or \texttt{None}, \texttt{sys.stdout} will be used. Since printed arguments are converted to text strings, \texttt{print()} cannot be used with binary mode file objects. For these, use \texttt{file.write(...)} instead.

Whether output is buffered is usually determined by \textit{file}, but if the
flush keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the flush keyword argument.

class property(fget=None, fset=None, fdel=None, doc=None)

Return a property attribute.

fget is a function for getting an attribute value. fset is a function for setting an attribute value. fdel is a function for deleting an attribute value. And doc creates a docstring for the attribute.

A typical use is to define a managed attribute x:

class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.

If c is an instance of C, c.x will invoke the getter, c.x = value will invoke the setter and del c.x the deleter.

If given, doc will be the docstring of the property attribute. Otherwise, the property will copy fget's docstring (if it exists). This makes it possible to create read-only properties easily using
property() as a decorator:

```python

class Parrot:
    def __init__(self):
        self._voltage = 100000

@property
def voltage(self):
    '''Get the current voltage.'''
    return self._voltage

The @property decorator turns the voltage() method into a “getter” for a read-only attribute with the same name, and it sets the docstring for voltage to “Get the current voltage.”

A property object has getter, setter, and deleter methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (\(x\) in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

*Changed in version 3.5*: The docstrings of property objects are now writeable.

```
def del self._x
```

### range

```
range(stop)
range(start, stop[, step])
```

Rather than being a function, `range` is actually an immutable sequence type, as documented in Ranges and Sequence Types — list, tuple, range.

### repr

```
repr(object)
```

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

### reversed

```
reversed(seq)
```

Return a reverse iterator. `seq` must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with
integer arguments starting at 0).

**round**(number[, ndigits])

Return number rounded to ndigits precision after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.

For the built-in types supporting round(), values are rounded to the closest multiple of 10 to the power minus ndigits; if two multiples are equally close, rounding is done toward the even choice (so, for example, both round(0.5) and round(-0.5) are 0, and round(1.5) is 2). Any integer value is valid for ndigits (positive, zero, or negative). The return value is an integer if ndigits is omitted or None. Otherwise the return value has the same type as number.

For a general Python object number, round delegates to number.__round__.

**Note:** The behavior of round() for floats can be surprising: for example, round(2.675, 2) gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See Floating Point Arithmetic: Issues and Limitations for more information.

**class set([iterable])**

Return a new set object, optionally with elements taken from iterable. set is a built-in class. See set and Set Types — set, frozenset for documentation about this class.
For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`setattr(object, name, value)`

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

`class slice(stop)`

`class slice(start, stop[, step])`

Return a `slice` object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator.

`sorted(iterable, *, key=None, reverse=False)`

Return a new sorted list from the items in `iterable`.

Has two optional arguments which must be specified as keyword arguments.

`key` specifies a function of one argument that is used to extract a comparison key from each element in `iterable` (for example,
key=str.lower). The default value is None (compare the elements directly).

reverse is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

Use functools.cmp_to_key() to convert an old-style cmp function to a key function.

The built-in sorted() function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

@staticmethod
Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C:
    @staticmethod
    def f(arg1, arg2, ...): ...

The @staticmethod form is a function decorator — see the description of function definitions in Function definitions for details.

It can be called either on the class (such as C.f()) or on an instance (such as C().f()). The instance is ignored except for its class.
Static methods in Python are similar to those found in Java or C++. Also see `classmethod()` for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```python
class C:
    builtin_open = staticmethod(open)
```

For more information on static methods, consult the documentation on the standard type hierarchy in *The standard type hierarchy*.

```python
class str(object='')
class str(object=b'', encoding='utf-8', errors='strict')
```

Return a `str` version of `object`. See `str()` for details.

`str` is the built-in string `class`. For general information about strings, see *Text Sequence Type — str*.

```python
sum(iterable[, start])
```

Sums `start` and the items of an `iterable` from left to right and returns the total. `start` defaults to 0. The `iterable`'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `'' . join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of
iterables, consider using `itertools.chain()`.

**super([type[, object-or-type]])**

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by `getattr()` except that the `type` itself is skipped.

The `__mro__` attribute of the `type` lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case
(because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```python
class C(B):
    def method(self, arg):
        super().method(arg)  # This does the same
        # super(C, self).method(arg)
```

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name].`

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see `guide to using super()`.

```python
tuple([iterable])
```
Rather than being a function, `tuple` is actually an immutable sequence type, as documented in Tuples and Sequence Types — list, tuple, range.

```python
class type(object)
class type(name, bases, dict)
```

With one argument, return the type of an `object`. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The `name` string is the class name and becomes the `__name__` attribute; the `bases` tuple itemizes the base classes and becomes the `__bases__` attribute; and the `dict` dictionary is the namespace containing definitions for class body and is copied to a standard dictionary to become the `__dict__` attribute. For example, the following two statements create identical `type` objects:

```python
>>> class X:
...   a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

See also Type Objects.

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.
**vars([object])**

Return the **__dict__** attribute for a module, class, instance, or any other object with a **__dict__** attribute.

Objects such as modules and instances have an updateable **__dict__** attribute; however, other objects may have write restrictions on their **__dict__** attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

**zip(*)iterables)**

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```python
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
```
The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the same iterator n times so that each output tuple has the result of n calls to the iterator. This has the effect of dividing the input into n-length chunks.

`zip()` should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```python
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

`__import__(name, globals=None, locals=None, fromlist=(), level=0)`

**Note:** This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.
This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see PEP 302) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module `name`, potentially using the given `globals` and `locals` to determine how to interpret the name in a package context. The `fromlist` gives the names of objects or submodules that should be imported from the module given by `name`. The standard implementation does not use its `locals` argument at all, and uses its `globals` only to determine the package context of the `import` statement.

`level` specifies whether to use absolute or relative imports. 0 (the default) means only perform absolute imports. Positive values for `level` indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see PEP 328 for the details).

When the `name` variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by `name`. However, when a non-empty `fromlist` argument is given, the module named by `name` is returned.

For example, the statement `import spam` results in bytecode resembling the following code:
The statement `import spam.ham` results in this call:

```python
spam = __import__('spam.ham', globals(), locals(), [])
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```python
_temp = __import__('spam.ham', globals(), locals())
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

*Changed in version 3.3*: Negative values for `level` are no longer supported (which also changes the default value to 0).

**Footnotes**

Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-
style newlines.
Built-in Constants

A small number of constants live in the built-in namespace. They are:

**False**
The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

**True**
The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

**None**
The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

**NotImplemented**
Special value which should be returned by the binary special methods (e.g. `__eq__`, `__lt__`, `__add__`, `__rsub__`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__`, `__iand__`, etc.) for the same purpose. Its truth value is true.

**Note:** When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter
will raise an appropriate exception. Incorrectly returning NotImplemented will result in a misleading error message or the NotImplemented value being returned to Python code. See Implementing the arithmetic operations for examples.

**Note:** NotImplemented and NotImplemented are not interchangeable, even though they have similar names and purposes. See NotImplemented for details on when to use it.

**Ellipsis**

The same as the ellipsis literal “...”. Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

**__debug__**

This constant is true if Python was not started with an -O option. See also the assert statement.

**Note:** The names None, False, True and __debug__ cannot be reassigned (assignments to them, even as an attribute name, raise SyntaxError), so they can be considered “true” constants.
Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

```python
quit(code=None)
exit(code=None)
```

Objects that when printed, print a message like “Use quit() or Ctrl-D (i.e. EOF) to exit”, and when called, raise `SystemExit` with the specified exit code.

```python
copyright
credits
```

Objects that when printed or called, print the text of copyright or credits, respectively.

```python
license
```

Object that when printed, prints the message “Type license() to see the full license text”, and when called, displays the full license text in a pager-like fashion (one screen at a time).
Built-in Types

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don’t return a specific item, never return the collection instance itself but None.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the \texttt{repr()} function or the slightly different \texttt{str()} function). The latter function is implicitly used when an object is written by the \texttt{print()} function.
Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object. [1] Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`.
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `'`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)
Boolean Operations — and, or, not

These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x or y</td>
<td>if x is false, then y, else x</td>
<td>(1)</td>
</tr>
<tr>
<td>x and y</td>
<td>if x is false, then x, else y</td>
<td>(2)</td>
</tr>
<tr>
<td>not x</td>
<td>if x is false, then True, else False</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.
Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, \( x < y <= z \) is equivalent to \( x < y \) and \( y <= z \), except that \( y \) is evaluated only once (but in both cases \( z \) is not evaluated at all when \( x < y \) is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>strictly less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>is</td>
<td>object identity</td>
</tr>
<tr>
<td>is not</td>
<td>negated object identity</td>
</tr>
</tbody>
</table>

Objects of different types, except different numeric types, never compare equal. Furthermore, some types (for example, function objects) support only a degenerate notion of comparison where any two objects of that type are unequal. The <, <=, > and >= operators will raise a `TypeError` exception when comparing a complex number with another built-in numeric type, when the objects are of different types that cannot be compared, or in other cases where there is no
defined ordering.

Non-identical instances of a class normally compare as non-equal unless the class defines the \_\_eq\_\_() method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods \_\_lt\_\_(), \_\_le\_\_(), \_\_gt\_\_(), and \_\_ge\_\_() (in general, \_\_lt\_\_() and \_\_eq\_\_() are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the \texttt{is} and \texttt{is not} operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, \texttt{in} and \texttt{not in}, are supported by types that are \texttt{iterable} or implement the \_\_contains\_\_() method.
Numeric Types — int, float, complex

There are three distinct numeric types: integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using double in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number \( z \), use \( z\).real and \( z\).imag. (The standard library includes additional numeric types, fractions that hold rationals, and decimal that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending 'j' or 'J' to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.
All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
<th>Full documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
<td>sum of $x$ and $y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x - y$</td>
<td>difference of $x$ and $y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x * y$</td>
<td>product of $x$ and $y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x / y$</td>
<td>quotient of $x$ and $y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \div y$</td>
<td>floored quotient of $x$ and $y$</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>$x % y$</td>
<td>remainder of $x / y$</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>-$x$</td>
<td>$x$ negated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+x$</td>
<td>$x$ unchanged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{abs}(x)$</td>
<td>absolute value or magnitude of $x$</td>
<td></td>
<td>$\text{abs}()$</td>
</tr>
<tr>
<td>$\text{int}(x)$</td>
<td>$x$ converted to integer</td>
<td>(3)(6)</td>
<td>$\text{int}()$</td>
</tr>
<tr>
<td>$\text{float}(x)$</td>
<td>$x$ converted to floating point</td>
<td>(4)(6)</td>
<td>$\text{float}()$</td>
</tr>
<tr>
<td>$\text{complex}(\text{re}, \text{im})$</td>
<td>a complex number with real part $\text{re}$, imaginary part $\text{im}$. $\text{im}$ defaults to zero.</td>
<td>(6)</td>
<td>$\text{complex}()$</td>
</tr>
<tr>
<td>$c\text{.conjugate()}$</td>
<td>conjugate of the complex number $c$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{divmod}(x, y)$</td>
<td>the pair $(x \div y, x % y)$</td>
<td>(2)</td>
<td>$\text{divmod}()$</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td><code>x</code> to the power <code>y</code></td>
<td>(5)</td>
<td></td>
</tr>
<tr>
<td><code>x ** y</code></td>
<td><code>x</code> to the power <code>y</code></td>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily int. The result is always rounded towards minus infinity: `1//2` is `0`, `(-1)//2` is `-1`, `1//(-2)` is `-1`, and `(-1)//(-2)` is `0`.

2. Not for complex numbers. Instead convert to floats using `abs()` if appropriate.

3. Conversion from floating point to integer may round or truncate as in C; see functions `math.floor()` and `math.ceil()` for well-defined conversions.

4. `float` also accepts the strings “nan” and “inf” with an optional prefix “+” or “-“ for Not a Number (NaN) and positive or negative infinity.

5. Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.

6. The numeric literals accepted include the digits `0` to `9` or any Unicode equivalent (code points with the Nd property).

See http://www.unicode.org/Public/10.0.0/ucd/extracted/DerivedNumericType.txt for a complete list of code points with the Nd property.

All `numbers.Real` types (int and float) also include the following operations:
### Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>math.trunc(x)</code></td>
<td><code>x</code> truncated to Integral</td>
</tr>
<tr>
<td><code>round(x[, n])</code></td>
<td><code>x</code> rounded to <code>n</code> digits, rounding half to even. If <code>n</code> is omitted, it defaults to 0.</td>
</tr>
<tr>
<td><code>math.floor(x)</code></td>
<td>the greatest Integral <code>&lt;= x</code></td>
</tr>
<tr>
<td><code>math.ceil(x)</code></td>
<td>the least Integral <code>&gt;= x</code></td>
</tr>
</tbody>
</table>

For additional numeric operations see the `math` and `cmath` modules.

### Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>`x</td>
<td>y`</td>
<td>bitwise or of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>x ^ y</code></td>
<td>bitwise exclusive or of <code>x</code> and <code>y</code></td>
<td>(4)</td>
</tr>
<tr>
<td><code>x &amp; y</code></td>
<td>bitwise and of <code>x</code> and <code>y</code></td>
<td>(4)</td>
</tr>
<tr>
<td><code>x &lt;&lt; n</code></td>
<td><code>x</code> shifted left by <code>n</code> bits</td>
<td>(1)(2)</td>
</tr>
<tr>
<td><code>x &gt;&gt; n</code></td>
<td><code>x</code> shifted right by <code>n</code> bits</td>
<td>(1)(3)</td>
</tr>
<tr>
<td><code>~x</code></td>
<td>the bits of <code>x</code> inverted</td>
<td></td>
</tr>
</tbody>
</table>
Notes:

1. Negative shift counts are illegal and cause a `ValueError` to be raised.
2. A left shift by \( n \) bits is equivalent to multiplication by \( \text{pow}(2, n) \) without overflow check.
3. A right shift by \( n \) bits is equivalent to division by \( \text{pow}(2, n) \) without overflow check.
4. Performing these calculations with at least one extra sign extension bit in a finite two's complement representation (a working bit-width of \( 1 + \max(x\text{.bit\_length}(), y\text{.bit\_length}()) \) or more) is sufficient to get the same result as if there were an infinite number of sign bits.

Additional Methods on Integer Types

The int type implements the `numbers.Integral abstract base class`. In addition, it provides a few more methods:

`int.bit\_length()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```python
g>>> n = -37
g>>> bin(n)
g'\-0b100101'
g>>> n.bit\_length()
g6
```

More precisely, if \( x \) is nonzero, then \( x\text{.bit\_length}() \) is the unique positive integer \( k \) such that \( 2^{(k-1)} \leq \text{abs}(x) < 2^k \). Equivalently, when \( \text{abs}(x) \) is small enough to have a correctly rounded logarithm, then \( k = 1 + \text{int}(\log(\text{abs}(x)), \)

2)). If x is zero, then `x.bit_length()` returns 0.

Equivalent to:

```python
def bit_length(self):
    s = bin(self)  # binary representation:
    s = s.lstrip('-0b')  # remove leading zeros and
    return len(s)  # len('100101') --> 6
```

*New in version 3.1.*

```
int.to_bytes(length, byteorder, *, signed=False)
```

Return an array of bytes representing an integer.

```python
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
\xff\xff\xff\xff\xff\xff\xff\xff\xc0'\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder)
\xe8\x03'
```

The integer is represented using `length` bytes. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

The `byteorder` argument determines the byte order used to represent the integer. If `byteorder` is "big", the most significant byte is at the beginning of the byte array. If `byteorder` is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as
the byte order value.

The *signed* argument determines whether two’s complement is used to represent the integer. If *signed* is `False` and a negative integer is given, an `OverflowError` is raised. The default value for *signed* is `False`.

*New in version 3.2.*

```
classmethod int.from_bytes(bytes, byteorder, *, signed=False)
```

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=-1024)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=64512)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a *bytes-like object* or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument indicates whether two’s complement is used
to represent the integer.

*New in version 3.2.*

## Additional Methods on Float

The float type implements the `numbers.Real` abstract base class. `float` also has the following additional methods.

**float.** `as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

**float.** `is_integer()`

Return `True` if the float instance is finite with integral value, and `False` otherwise:

```python
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a `decimal` string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

**float.** `hex()`

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this
representation will always include a leading 0x and a trailing p and exponent.

classmethod float.fromhex(s)
Class method to return the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

Note that float.hex() is an instance method, while float.fromhex() is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.'] fraction] ['p'] exponent
```

where the optional sign may by either + or -, integer and fraction are strings of hexadecimal digits, and exponent is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of float.hex() is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's %a format character or Java's Double.toHex() are accepted by float.fromhex().

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string 0x3.a7p10 represents the floating-point number \((3 + 10./16 + 7./16**2) \times 2.0**10\), or 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```
Applying the reverse conversion to 3740.0 gives a different hexadecimal string representing the same number:

```python
g>>> float.hex(3740.0)
'0x1.d3800000000000p+11'
```

Hashing of numeric types

For numbers x and y, possibly of different types, it's a requirement that hash(x) == hash(y) whenever x == y (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo P for a fixed prime P. The value of P is made available to Python as the modulus attribute of `sys.hash_info`.

**CPython implementation detail:** Currently, the prime used is P = $2^{31} - 1$ on machines with 32-bit C longs and P = $2^{61} - 1$ on machines with 64-bit C longs.

Here are the rules in detail:

- If $x = \frac{m}{n}$ is a nonnegative rational number and n is not divisible by P, define hash(x) as $m \times \text{invmod}(n, P) \mod P$, where `invmod(n, P)` gives the inverse of n modulo P.
- If $x = \frac{m}{n}$ is a nonnegative rational number and n is divisible
by \( P \) (but \( m \) is not) then \( n \) has no inverse modulo \( P \) and the rule above doesn't apply; in this case define \( \text{hash}(x) \) to be the constant value \( \text{sys.hash_info.inf} \).

- If \( x = m / n \) is a negative rational number define \( \text{hash}(x) \) as \(-\text{hash}(-x)\). If the resulting hash is \(-1\), replace it with \(-2\).
- The particular values \( \text{sys.hash_info.inf} \), \(-\text{sys.hash_info.inf}\) and \( \text{sys.hash_info.nan} \) are used as hash values for positive infinity, negative infinity, or nans (respectively). (All hashable nans have the same hash value.)
- For a \textbf{complex} number \( z \), the hash values of the real and imaginary parts are combined by computing \( \text{hash}(z.\text{real}) + \text{sys.hash_info.imag} \times \text{hash}(z.\text{imag}) \), reduced modulo \( 2^{\text{sys.hash_info.width}} \) so that it lies in range\((-2^{\text{sys.hash_info.width} - 1}, 2^{\text{sys.hash_info.width} - 1})\). Again, if the result is \(-1\), it's replaced with \(-2\).

To clarify the above rules, here's some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, \texttt{float}, or \texttt{complex}:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number \( m / n \).

    Assumes \( m \) and \( n \) are integers, with \( n \) positive. Equivalent to \text{hash}(\text{fractions.Fraction}(m, n)).
    """

    P = sys.hash_info.modulus
    # Remove common factors of \( P \). (Unnecessary if \( m \) and \( n \) are coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P
```
if n % P == 0:
    hash_value = sys.hash_info.inf
else:
    # Fermat's Little Theorem: pow(n, P-1, P) is 1,
    # so pow(n, P-2, P) gives the inverse of n modulo P.
    hash_value = (abs(m) % P) * pow(n, P - 2, P)
if m < 0:
    hash_value = -hash_value
if hash_value == -1:
    hash_value = -2
return hash_value

def hash_float(x):
    """Compute the hash of a float x."""
    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""
    hash_value = hash_float(z.real) + sys.hash_info.inf
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value:
    if hash_value == -1:
        hash_value = -2
    return hash_value
Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

```
container.__iter__()  
```

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

```
iterator.__iter__()  
```

Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

```
iterator.__next__()  
```

Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method
corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator's `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

**Generator Types**

Python's *generators* provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the `yield` expression.
Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The collections.abc.Sequence ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, s and t are sequences of the same type, n, i, j and k are integers and x is an arbitrary object that meets any type and value restrictions imposed by s.

The in and not in operations have the same priorities as the comparison operations. The + (concatenation) and * (repetition) operations have the same priority as the corresponding numeric operations. [3]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>True if an item of s is equal to x, else False</td>
<td>(1)</td>
</tr>
<tr>
<td>x not in s</td>
<td>False if an item of s is equal to x, else True</td>
<td>(1)</td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
<td>(6)(7)</td>
</tr>
<tr>
<td>s * n or n * s</td>
<td>equivalent to adding s to itself</td>
<td>(2)(7)</td>
</tr>
</tbody>
</table>
### Shortcuts

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[i]</code></td>
<td><code>i</code>th item of <code>s</code>, origin 0</td>
<td>(3)</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code></td>
<td>(3)(4)</td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code></td>
<td>(3)(5)</td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>smallest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>largest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>s.index(x[, i[, j]])</code></td>
<td>index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)</td>
<td>(8)</td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>total number of occurrences of <code>x</code> in <code>s</code></td>
<td></td>
</tr>
</tbody>
</table>

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see Comparisons in the language reference.)

**Notes:**

1. While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as `str`, `bytes` and `bytearray`) also use them for subsequence testing:

```python
>>> "gg" in "eggs"
True
```
2. Values of \( n \) less than 0 are treated as 0 (which yields an empty sequence of the same type as \( s \)). Note that items in the sequence \( s \) are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```python
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that \([[]\) is a one-element list containing an empty list, so all three elements of \([[[]\] * 3 are references to this single empty list. Modifying any of the elements of \( \text{lists} \) modifies this single list. You can create a list of different lists this way:

```python
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry How do I create a multidimensional list?.

3. If \( i \) or \( j \) is negative, the index is relative to the end of sequence \( s \): \( \text{len}(s) + i \) or \( \text{len}(s) + j \) is substituted. But note that \(-0\) is still 0.

4. The slice of \( s \) from \( i \) to \( j \) is defined as the sequence of items with index \( k \) such that \( i \leq k < j \). If \( i \) or \( j \) is greater than \( \text{len}(s) \),
use `len(s)`. If `i` is omitted or `None`, use `0`. If `j` is omitted or `None`, use `len(s)`. If `i` is greater than or equal to `j`, the slice is empty.

5. The slice of `s` from `i` to `j` with step `k` is defined as the sequence of items with index \( x = i + n \times k \) such that \( 0 \leq n < (j-i)/k \). In other words, the indices are `i, i+k, i+2*k, i+3*k` and so on, stopping when `j` is reached (but never including `j`). When `k` is positive, `i` and `j` are reduced to `len(s)` if they are greater. When `k` is negative, `i` and `j` are reduced to `len(s) - 1` if they are greater. If `i` or `j` are omitted or `None`, they become “end” values (which end depends on the sign of `k`). Note, `k` cannot be zero. If `k` is `None`, it is treated like `1`.

6. Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:

- if concatenating `str` objects, you can build a list and use `str.join()` at the end or else write to an `io.StringIO` instance and retrieve its value when complete
- if concatenating `bytes` objects, you can similarly use `bytes.join()` or `io.BytesIO`, or you can do in-place concatenation with a `bytearray` object. `bytearray` objects are mutable and have an efficient overallocation mechanism
- if concatenating `tuple` objects, extend a `list` instead
- for other types, investigate the relevant class documentation

7. Some sequence types (such as `range`) only support item sequences that follow specific patterns, and hence don’t support sequence concatenation or repetition.
8. `index` raises `ValueError` when `x` is not found in `s`. Not all implementations support passing the additional arguments `i` and `j`. These arguments allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using `s[i:j].index(x)`, only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

This support allows immutable sequences, such as `tuple` instances, to be used as `dict` keys and stored in `set` and `frozenset` instances.

Attempting to hash an immutable sequence that contains unhashable values will result in `TypeError`.

Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The `collections.abc.MutableSequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table `s` is an instance of a mutable sequence type, `t` is any iterable object and `x` is an arbitrary object that meets any type and value restrictions imposed by `s` (for example, `bytearray` only
accepts integers that meet the value restriction \(0 \leq x \leq 255\).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s[i] = x)</td>
<td>item (i) of (s) is replaced by (x)</td>
<td></td>
</tr>
<tr>
<td>(s[i:j] = t)</td>
<td>slice of (s) from (i) to (j) is replaced by the contents of the iterable (t)</td>
<td></td>
</tr>
<tr>
<td>(\text{del } s[i:j])</td>
<td>same as (s[i:j] = [])</td>
<td></td>
</tr>
<tr>
<td>(s[i:j:k] = t)</td>
<td>the elements of (s[i:j:k]) are replaced by those of (t)</td>
<td>(1)</td>
</tr>
<tr>
<td>(\text{del } s[i:j:k])</td>
<td>removes the elements of (s[i:j:k]) from the list</td>
<td></td>
</tr>
<tr>
<td>(s.append(x))</td>
<td>appends (x) to the end of the sequence (same as (s[len(s):len(s)] = [x]))</td>
<td></td>
</tr>
<tr>
<td>(s.clear())</td>
<td>removes all items from (s) (same as (\text{del } s[:]))</td>
<td>(5)</td>
</tr>
<tr>
<td>(s.copy())</td>
<td>creates a shallow copy of (s) (same as (s[:]))</td>
<td>(5)</td>
</tr>
<tr>
<td>(s.extend(t)) or (s += t)</td>
<td>extends (s) with the contents of (t) (for the most part the same as (s[len(s):len(s)] = t))</td>
<td></td>
</tr>
<tr>
<td>(s *= n)</td>
<td>updates (s) with its contents repeated (n) times</td>
<td>(6)</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>s.insert(i, x)</code></td>
<td>inserts <code>x</code> into <code>s</code> at the index given by <code>i</code> (same as <code>s[i:i] = [x]</code>)</td>
<td></td>
</tr>
<tr>
<td><code>s.pop([i])</code></td>
<td>retrieves the item at <code>i</code> and also removes it from <code>s</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td>remove the first item from <code>s</code> where <code>s[i]</code> is equal to <code>x</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td><code>s.reverse()</code></td>
<td>reverses the items of <code>s</code> in place</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. `t` must have the same length as the slice it is replacing.

2. The optional argument `i` defaults to `-1`, so that by default the last item is removed and returned.

3. `remove` raises `ValueError` when `x` is not found in `s`.

4. The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.

5. `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don’t support slicing operations (such as `dict` and `set`)

   *New in version 3.3:* `clear()` and `copy()` methods.

6. The value `n` is an integer, or an object implementing
Zero and negative values of $n$ clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for $s^{n}$ under Common Sequence Operations.

Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

```python
class list([iterable])
```

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a], [a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned, similar to `iterable[:].` For example, `list('abc')` returns `['a', 'b', 'c']` and `list((1, 2, 3))` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[ ]`.

Many other operations also produce lists, including the `sorted()` built-in.

Lists implement all of the common and mutable sequence operations. Lists also provide the following additional method:
**sort(***, **key=None, reverse=False)**

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

**sort()** accepts two arguments that can only be passed by keyword (**keyword-only arguments**):

- **key** specifies a function of one argument that is used to extract a comparison key from each list element (for example, **key=str.lower**). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of **None** means that list items are sorted directly without calculating a separate key value.

  The **functools.cmp_to_key()** utility is available to convert a 2.x style **cmp** function to a **key** function.

- **reverse** is a boolean value. If set to **True**, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use **sorted()** to explicitly request a new sorted list instance).

The **sort()** method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).
CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.

Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the `enumerate()` built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a `set` or `dict` instance).

```python
class tuple(iterable)
```

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the `tuple()` built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a tuple, it is returned unchanged. For example, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, the constructor creates a new empty tuple, `()`.

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty
tupled tuple case, or when they are needed to avoid syntactic ambiguity. For example, \( f(a, b, c) \) is a function call with three arguments, while \( f((a, b, c)) \) is a function call with a 3-tuple as the sole argument.

Tuples implement all of the common sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, `collections.namedtuple()` may be a more appropriate choice than a simple tuple object.

**Ranges**

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

```python
class range(stop)
class range(start, stop[, step])
```

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the `step` argument is omitted, it defaults to 1. If the `start` argument is omitted, it defaults to 0. If `step` is zero, `ValueError` is raised.

For a positive `step`, the contents of a range \( r \) are determined by the formula \( r[i] = \text{start} + \text{step} \times i \) where \( i \geq 0 \) and \( r[i] < \text{stop} \).

For a negative `step`, the contents of the range are still determined by the formula \( r[i] = \text{start} + \text{step} \times i \), but the constraints are \( i \geq 0 \) and \( r[i] > \text{stop} \).
A range object will be empty if \( r[0] \) does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than \( \text{sys.maxsize} \) are permitted but some features (such as \( \text{len()} \)) may raise \texttt{OverflowError}.

Range examples:

```python
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the \texttt{common} sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

**start**

The value of the \texttt{start} parameter (or 0 if the parameter was not supplied)
**stop**

The value of the *stop* parameter

**step**

The value of the *step* parameter (or 1 if the parameter was not supplied)

The advantage of the *range* type over a regular *list* or *tuple* is that a *range* object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the *start*, *stop* and *step* values, calculating individual items and subranges as needed).

Range objects implement the *collections.abc.Sequence* ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see *Sequence Types — list, tuple, range*):

```python
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```
Testing range objects for equality with == and != compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different start, stop and step attributes, for example range(0) == range(2, 1, 3) or range(0, 3, 2) == range(0, 4, 2).)

*Changed in version 3.2:* Implement the Sequence ABC. Support slicing and negative indices. Test int objects for membership in constant time instead of iterating through all items.

*Changed in version 3.3:* Define ‘==’ and ‘!=' to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

*New in version 3.3:* The start, stop and step attributes.

**See also:**

- The linspace recipe shows how to implement a lazy version of range suitable for floating point applications.
Text Sequence Type — **str**

Textual data in Python is handled with *str* objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: "allows embedded 'single' quotes". 
- Triple quoted: """Three single quotes"", """"Three double quotes""

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, ("spam " "eggs") == "spam eggs".

See **String and Bytes literals** for more about the various forms of string literal, including supported escape sequences, and the `r` ("raw") prefix that disables most escape sequence processing.

Strings may also be created from other objects using the *str* constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string *s*, *s[0] == s[0:1]*.

There is also no mutable string type, but *str.join()* or *io.StringIO* can be used to efficiently construct strings from multiple fragments.
Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

```python
class str(object="")
class str(object=b'', encoding='utf-8', errors='strict')
```

Return a string version of object. If object is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether `encoding` or `errors` is given, as follows.

If neither `encoding` nor `errors` is given, `str(object)` returns `object.__str__()`, which is the “informal” or nicely printable string representation of `object`. For string objects, this is the string itself. If `object` does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

If at least one of `encoding` or `errors` is given, `object` should be a bytes-like object (e.g. `bytes` or `bytearray`). In this case, if `object` is a `bytes` (or `bytearray`) object, then `str(bytes, encoding, errors)` is equivalent to `bytes.decode(encoding, errors)`. Otherwise, the bytes object underlying the buffer object is obtained before calling `bytes.decode()`. See Binary Sequence Types — `bytes`, `bytearray`, `memoryview` and Buffer Protocol for information on buffer objects.

Passing a `bytes` object to `str()` without the `encoding` or `errors` arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:
For more information on the `str` class and its methods, see Text Sequence Type — str and the String Methods section below. To output formatted strings, see the Formatted string literals and Format String Syntax sections. In addition, see the Text Processing Services section.

**String Methods**

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`), Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

**str.capitalize()**
Return a copy of the string with its first character capitalized and the rest lowercased.

**str.casefold()**
Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.
Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to 'ß'; `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

*New in version 3.3.*

```
str.center(width[, fillchar])
```

Return centered in a string of length `width`. Padding is done using the specified `fillchar` (default is an ASCII space). The original string is returned if `width` is less than or equal to `len(s)`.

```
str.count(sub[, start[, end]])
```

Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

```
str.encode(encoding="utf-8", errors="strict")
```

Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. `errors` may be given to set a different error handling scheme. The default for `errors` is 'strict', meaning that encoding errors raise a `UnicodeError`. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via `codecs.register_error()`, see section Error Handlers. For a list of possible encodings, see section Standard Encodings.

*Changed in version 3.1:* Support for keyword arguments added.
str.\texttt{endswith}(\texttt{suffix[, start[, end]])}

Return \texttt{True} if the string ends with the specified \texttt{suffix}, otherwise return \texttt{False}. \texttt{suffix} can also be a tuple of suffixes to look for. With optional \texttt{start}, test beginning at that position. With optional \texttt{end}, stop comparing at that position.

str.\texttt{expandtabs}(\texttt{tabsize=8})

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every \texttt{tabsize} characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\texttt{\textbackslash t}), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\texttt{\textbackslash n}) or return (\texttt{\textbackslash r}), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

\begin{verbatim}
>>> '01\t012\t0123\t01234'.expandtabs()
'01    012   0123   01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01 012 0123 01234'
\end{verbatim}

str.\texttt{find}(\texttt{sub[, start[, end]])}

Return the lowest index in the string where substring \texttt{sub} is found within the slice \texttt{s[start:end]}. Optional arguments \texttt{start} and \texttt{end} are interpreted as in slice notation. Return \texttt{-1} if \texttt{sub} is not found.

\textbf{Note:} The \texttt{find()} method should be used only if you need to
know the position of sub. To check if sub is a substring or not, use the in operator:

```python
>>> 'Py' in 'Python'
True
```

**str.format(*)*args, **kwargs**

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```python
>>> "The sum of 1 + 2 is \{0\}\".format(1+2)
'The sum of 1 + 2 is 3'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

**Note:** When formatting a number (int, float, complex, decimal.Decimal and subclasses) with the n type (ex: '{:n}'.format(1234)), the function temporarily sets the LC_CTYPE locale to the LC_NUMERIC locale to decode decimal_point and thousands_sep fields of localeconv() if they are non-ASCII or longer than 1 byte, and the LC_NUMERIC locale is different than the LC_CTYPE locale. This temporary change affects other threads.

*Changed in version 3.7:* When formatting a number with the n type, the function sets temporarily the LC_CTYPE locale to the
LC_NUMERIC locale in some cases.

str.format_map(mapping)
Similar to str.format(**mapping), except that mapping is used directly and not copied to a dict. This is useful if for example mapping is a dict subclass:

```python
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default('Guido was born in country'))
'Guido was born in country'
```

New in version 3.2.

str.index(sub[, start[, end]])
Like find(), but raise ValueError when the substring is not found.

str.isalnum()
Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise. A character c is alphanumeric if one of the following returns True: c.isalpha(), c.isdecimal(), c.isdigit(), or c.isnumeric().

str.isalpha()
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the
“Alphabetic” property defined in the Unicode Standard.

str.isascii()
    Return true if the string is empty or all characters in the string are ASCII, false otherwise. ASCII characters have code points in the range U+0000-U+007F.

    New in version 3.7.

str.isdecimal()
    Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category “Nd”.

str.isdigit()
    Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.isidentifier()
    Return true if the string is a valid identifier according to the language definition, section Identifiers and keywords.

    Use keyword.iskeyword() to test for reserved identifiers such as def and class.

str.islower()
Return true if all cased characters [4] in the string are lowercase and there is at least one cased character, false otherwise.

`str.isnumeric()`  
Return true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

`str.isprintable()`  
Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

`str.isspace()`  
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as “Other” or “Separator” and those with bidirectional property being one of “WS”, “B”, or “S”.

`str.istitle()`  
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow
uncased characters and lowercase characters only cased ones. Return false otherwise.

str.isupper()
Return true if all cased characters [4] in the string are uppercase and there is at least one cased character, false otherwise.

str.join(iterable)
Return a string which is the concatenation of the strings in iterable. A TypeError will be raised if there are any non-string values in iterable, including bytes objects. The separator between elements is the string providing this method.

str.ljust(width[, fillchar])
Return the string left justified in a string of length width. Padding is done using the specified fillchar (default is an ASCII space). The original string is returned if width is less than or equal to len(s).

str.lower()
Return a copy of the string with all the cased characters [4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

str.lstrip([chars])
Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped:
>>> ' spacious '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')</a>
'exmaple.com'

```
static str.maketrans(x[, y[, z]])
```

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

```
str.partition(sep)
```

Split the string at the first occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

```
str.replace(old, new[, count])
```

Return a copy of the string with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.

```
str.rfind(sub[, start[, end]])
```
Return the highest index in the string where substring \textit{sub} is found, such that \textit{sub} is contained within \texttt{s[start:end]}. Optional arguments \texttt{start} and \texttt{end} are interpreted as in slice notation. Return \texttt{-1} on failure.

\texttt{str.rindex(sub[, start[, end]])}

Like \texttt{rfind()} but raises \texttt{ValueError} when the substring \textit{sub} is not found.

\texttt{str.rjust(width[, fillchar])}

Return the string right justified in a string of length \textit{width}. Padding is done using the specified \textit{fillchar} (default is an ASCII space). The original string is returned if \textit{width} is less than or equal to \texttt{len(s)}.

\texttt{str.rpartition(sep)}

Split the string at the last occurrence of \textit{sep}, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

\texttt{str.rsplit(sep=None, maxsplit=-1)}

Return a list of the words in the string, using \textit{sep} as the delimiter string. If \textit{maxsplit} is given, at most \textit{maxsplit} splits are done, the \textit{rightmost} ones. If \textit{sep} is not specified or \texttt{None}, any whitespace string is a separator. Except for splitting from the right, \texttt{rsplit()} behaves like \texttt{split()} which is described in detail below.

\texttt{str.rstrip([chars])}

Return a copy of the string with trailing characters removed. The \textit{chars} argument is a string specifying the set of characters to be removed. If omitted or \texttt{None}, the \textit{chars} argument defaults to
removing whitespace. The chars argument is not a suffix; rather, all combinations of its values are stripped:

```python
>>> ' spacious '.rstrip()
' spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

**str.split(sep=None, maxsplit=-1)**

Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns [''].

For example:

```python
>>> '1,2,3'.split(',')
[1, '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
[1, '2,3']
>>> '1,2,,3,'.split(',
[1, '2', '', '3', '']
```

If sep is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single
separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```python
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

`str.splitlines([keepends])`  
Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless `keepends` is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of universal newlines.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Line Feed</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>\r\n</td>
<td>Carriage Return + Line Feed</td>
</tr>
<tr>
<td>\v or \x0b</td>
<td>Line Tabulation</td>
</tr>
<tr>
<td>\f or \x0c</td>
<td>Form Feed</td>
</tr>
<tr>
<td>\x1c</td>
<td>File Separator</td>
</tr>
<tr>
<td>\x1d</td>
<td>Group Separator</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>\x1e</td>
<td>Record Separator</td>
</tr>
<tr>
<td>\x85</td>
<td>Next Line (C1 Control Code)</td>
</tr>
<tr>
<td>\u2028</td>
<td>Line Separator</td>
</tr>
<tr>
<td>\u2029</td>
<td>Paragraph Separator</td>
</tr>
</tbody>
</table>

*Changed in version 3.2:* \v and \f added to list of line boundaries.

For example:

```python
>>> 'ab c
nde fg
kl
'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c
nde fg
kl
'.splitlines(keepends=True)
['ab c
', '\n', 'de fg\r', 'kl\r\n']
```

Unlike `split()` when a delimiter string `sep` is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```python
>>> ''.splitlines()
[]
>>> 'One line
'.splitlines()
['One line']
```

For comparison, `split('\n')` gives:

```python
>>> ''
['']
>>> 'Two lines
'.split('\n')
['Two lines', '']
```

```python
str.startswith(prefix[, start[, end]])
```

Return `True` if string starts with the `prefix`, otherwise return
False. `prefix` can also be a tuple of prefixes to look for. With optional `start`, test string beginning at that position. With optional `end`, stop comparing string at that position.

```python
>>> comment_string = '#........ Section 3.2.1 Issue #32
>>> comment_string.strip('#!')
'Section 3.2.1 Issue #32'
```

**str.strip([chars])**

Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```python
>>> ' spacious '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing `chars` argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in `chars`. A similar action takes place on the trailing end. For example:

```python
>>> comment_string = '#........ Section 3.2.1 Issue #32
>>> comment_string.strip('#!')
'Section 3.2.1 Issue #32'
```

**str.swapcase()**

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

**str.title()**
Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

```python
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```python
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```python
>>> import re

>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() + mo.group(0)[1:].lower(),
...                   s)

>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.translate(table)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an
object that implements indexing via `__getitem__()`, typically a mapping or sequence. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a `LookupError` exception, to map the character to itself.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

See also the `codecs` module for a more flexible approach to custom character mappings.

**`str.upper()`**
Return a copy of the string with all the cased characters converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

**`str.zfill(width)`**
Return a copy of the string left filled with ASCII '0' digits to make a string of length `width`. A leading sign prefix ('+'/'-') is handled by inserting the padding after the sign character rather than before. The original string is returned if `width` is less than or equal to `len(s)`.

For example:
printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals, the `str.format()` interface, or template strings may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string formatting or interpolation operator. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to using the `sprintf()` in the C language.

If `format` requires a single argument, `values` may be a single non-tuple object. Otherwise, `values` must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `%` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence
of characters (for example, (somename)).

3. Conversion flags (optional), which affect the result of some conversion types.

4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.

5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual precision is read from the next element of the tuple in values, and the value to convert comes after the precision.


7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

```python
>>> print('%(language)s has %(number)03d quote types
...   { 'language': "Python", "number": 2 })
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'!'</td>
<td>The value conversion will use the “alternate form” (where defined below).</td>
</tr>
</tbody>
</table>
The conversion will be zero padded for numeric values.

The converted value is left adjusted (overrides the '0' conversion if both are given).

(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

A sign character ('+' or '-' ) will precede the conversion (overrides a “space” flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td>Signed octal value.</td>
<td></td>
</tr>
<tr>
<td>'u'</td>
<td>Obsolete type – it is identical to 'd'.</td>
<td>(6)</td>
</tr>
<tr>
<td>'x'</td>
<td>Signed hexadecimal (lowercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>'X'</td>
<td>Signed hexadecimal (uppercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>'e'</td>
<td>Floating point exponential format (lowercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>'E'</td>
<td>Floating point exponential format (uppercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>'f'</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>'F'</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>'g'</td>
<td>Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format</td>
<td>(4)</td>
</tr>
</tbody>
</table>
Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise. (4)

Single character (accepts integer or single character string). (4)

String (converts any Python object using repr()). (5)

String (converts any Python object using str()). (5)

String (converts any Python object using ascii()). (5)

No argument is converted, results in a '%' character in the result. (4)

Notes:

1. The alternate form causes a leading octal specifier ('\o') to be inserted before the first digit.

2. The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.

3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.

4. The alternate form causes the result to always contain a decimal point.
point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. If precision is \( N \), the output is truncated to \( N \) characters.

6. See PEP 237.

Since Python strings have an explicit length, \( \%s \) conversions do not assume that '\0' is the end of the string.

*Changed in version 3.1:* \( \%f \) conversions for numbers whose absolute value is over 1e50 are no longer replaced by \( \%g \) conversions.
Binary Sequence Types — bytes, bytearray, memoryview

The core built-in types for manipulating binary data are bytes and bytearray. They are supported by memoryview which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The array module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

```python
class bytes([source[, encoding[, errors]]])
```

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a \texttt{b} prefix is added:

- Single quotes: \texttt{b}'still allows embedded "double" quotes'\texttt{'}
- Double quotes: \texttt{b"still allows embedded 'single' quotes"}.
- Triple quoted: \texttt{b''3 single quotes''}, \texttt{b"""3 double quotes""""}

Only ASCII characters are permitted in bytes literals (regardless of
the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a \r prefix to disable processing of escape sequences. See String and Bytes literals for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger ValueError). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: \texttt{bytes(10)}
- From an iterable of integers: \texttt{bytes(range(20))}
- Copying existing binary data via the buffer protocol: \texttt{bytes(obj)}

Also see the \texttt{bytes} built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:
classmethod `fromhex(string)`

This `bytes` class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```python
>>> bytes.fromhex('2Ef0 F1f2 ')
'b\.xf0\xf1\xf2'
```

*Changed in version 3.7:* `bytes.fromhex()` now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

`hex()`

Return a string object containing two hexadecimal digits for each byte in the instance.

```python
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

*New in version 3.5.*

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`. 
Note: For Python 2.x users: In the Python 2.x series, a variety of implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings were permitted. This was a backwards compatibility workaround to account for the fact that Python originally only supported 8-bit text, and Unicode text was a later addition. In Python 3.x, those implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and bytes and string objects will always compare unequal.

Bytearray Objects

bytearray objects are a mutable counterpart to bytes objects.

class bytearray([source[, encoding[, errors]]])

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: bytearray()
- Creating a zero-filled instance with a given length: bytearray(10)
- From an iterable of integers: bytearray(range(20))
- Copying existing binary data via the buffer protocol: bytearray(b'Hi!')

As bytearray objects are mutable, they support the mutable sequence operations in addition to the common bytes and bytearray operations described in Bytes and Bytearray Operations.

Also see the bytearray built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing
binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

**class method fromhex(string)**

This *bytearray* class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```python
>>> bytearray.fromhex('2Ef0 F1f2 ')
bytearray(b'\xf0\xf1\xf2')
```

*Changed in version 3.7:* *bytearray.fromhex()* now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

**hex()**

Return a string object containing two hexadecimal digits for each byte in the instance.

```python
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

*New in version 3.5.*

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object *b*, *b[0]* will be an integer, while *b[0:1]* will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format
(bytearray(b'...')) since it is often more useful than e.g. bytearray([46, 46, 46]). You can always convert a bytearray object into a list of integers using list(b).

Bytes and Bytearray Operations

Both bytes and bytearray objects support the common sequence operations. They interoperate not just with operands of the same type, but with any bytes-like object. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

**Note:** The methods on bytes and bytearray objects don’t accept strings as their arguments, just as the methods on strings don’t accept bytes as their arguments. For example, you have to write:

```python
a = "abc"
b = a.replace("a", "f")
```

and:

```python
a = b"abc"
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

**Note:** Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.
The following methods on bytes and bytearray objects can be used with arbitrary binary data.

```python
bytes.count(sub[, start[, end]])
bytearray.count(sub[, start[, end]])
```

Return the number of non-overlapping occurrences of subsequence `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

*Changed in version 3.3:* Also accept an integer in the range 0 to 255 as the subsequence.

```python
bytes.decode(encoding="utf-8", errors="strict")
bytearray.decode(encoding="utf-8", errors="strict")
```

Return a string decoded from the given bytes. Default encoding is 'utf-8'. `errors` may be given to set a different error handling scheme. The default for `errors` is 'strict', meaning that encoding errors raise a `UnicodeError`. Other possible values are 'ignore', 'replace' and any other name registered via `codecs.register_error()`, see section Error Handlers. For a list of possible encodings, see section Standard Encodings.

**Note:** Passing the `encoding` argument to `str` allows decoding any bytes-like object directly, without needing to make a temporary bytes or bytearray object.

*Changed in version 3.1:* Added support for keyword arguments.
bytes.\textbf{endswith}\((\text{suffix}\,[\,\text{start}[\,\text{end}]]\))
bytearray.\textbf{endswith}\((\text{suffix}\,[\,\text{start}[\,\text{end}]]\))

Return \textbf{True} if the binary data ends with the specified \textit{suffix},
otherwise return \textbf{False}. \textit{suffix} can also be a tuple of suffixes to
look for. With optional \textit{start}, test beginning at that position. With
optional \textit{end}, stop comparing at that position.

The suffix(es) to search for may be any \textit{bytes-like object}.

bytes.\textbf{find}\((\text{sub}\,[\,\text{start}[\,\text{end}]]\))
bytearray.\textbf{find}\((\text{sub}\,[\,\text{start}[\,\text{end}]]\))

Return the lowest index in the data where the subsequence \textit{sub} is
found, such that \textit{sub} is contained in the slice \texttt{s[start:end]}.
Optional arguments \textit{start} and \textit{end} are interpreted as in slice
notation. Return \texttt{-1} if \textit{sub} is not found.

The subsequence to search for may be any \textit{bytes-like object} or an
integer in the range 0 to 255.

\begin{mdframed}
\textbf{Note:} The \texttt{find()} method should be used only if you need to
know the position of \textit{sub}. To check if \textit{sub} is a substring or not,
use the \texttt{in} operator:
\begin{verbatim}
>>> b'Py' in b'Python'
True
\end{verbatim}
\end{mdframed}

\textit{Changed in version 3.3:} Also accept an integer in the range 0 to
255 as the subsequence.

bytes.\textbf{index}\((\text{sub}\,[\,\text{start}[\,\text{end}]]\))
bytearray.\textbf{index}\((\text{sub}\,[\,\text{start}[\,\text{end}]]\))
Like `find()`, but raise `ValueError` when the subsequence is not found.

The subsequence to search for may be any `bytes-like object` or an integer in the range 0 to 255.

*Changed in version 3.3:* Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.join(iterable)`  
`bytearray.join(iterable)`  
Return a bytes or bytearray object which is the concatenation of the binary data sequences in `iterable`. A `TypeError` will be raised if there are any values in `iterable` that are not `bytes-like objects`, including `str` objects. The separator between elements is the contents of the bytes or bytearray object providing this method.

`static bytes.maketrans(from, to)`  
`static bytearray.maketrans(from, to)`  
This static method returns a translation table usable for `bytes.translate()` that will map each character in `from` into the character at the same position in `to`; `from` and `to` must both be `bytes-like objects` and have the same length.

*New in version 3.1.*

`bytes.partition(sep)`  
`bytearray.partition(sep)`  
Split the sequence at the first occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray
The separator to search for may be any bytes-like object.

bytes.replace(old, new[, count])
byterray.replace(old, new[, count])

Return a copy of the sequence with all occurrences of subsequence old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

The subsequence to search for and its replacement may be any bytes-like object.

Note: The bytearray version of this method does not operate in place - it always produces a new object, even if no changes were made.

bytes.rfind(sub[, start[, end]])
byterray.rfind(sub[, start[, end]])

Return the highest index in the sequence where the subsequence sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.rindex(sub[, start[, end]])
byterray.rindex(sub[, start[, end]])
Like `rfind()` but raises `ValueError` when the subsequence `sub` is not found.

The subsequence to search for may be any `bytes-like object` or an integer in the range 0 to 255.

*Changed in version 3.3:* Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rpartition(sep)`
`bytestring.rpartition(sep)`
Split the sequence at the last occurrence of `sep`, and return a 3-tuple containing the part before the separator, the separator itself or its bytestring copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytestring objects.

The separator to search for may be any `bytes-like object`.

`bytes.startswith(prefix[, start[, end]])`
`bytestring.startswith(prefix[, start[, end]])`
Return `True` if the binary data starts with the specified `prefix`, otherwise return `False`. `prefix` can also be a tuple of prefixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position.

The prefix(es) to search for may be any `bytes-like object`.

`bytes.translate(table, delete=b")`  
`bytestring.translate(table, delete=b")`
Return a copy of the bytes or bytestring object where all bytes occurring in the optional argument `delete` are removed, and the
remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the `table` argument to `None` for translations that only delete characters:

```python
g>>> b'read this short text'.translate(None, b'aeiou')
gb'rd ths shrt txt'
```

*Changed in version 3.6:* `delete` is now supported as a keyword argument.

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

```python
bytes.center(width[, fillbyte])
bytearray.center(width[, fillbyte])
```

Return a copy of the object centered in a sequence of length `width`. Padding is done using the specified `fillbyte` (default is an ASCII space). For `bytes` objects, the original sequence is returned if `width` is less than or equal to `len(s)`.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.
bytes.\texttt{ljust}(width[, fillbyte])
bytearray.\texttt{ljust}(width[, fillbyte])

Return a copy of the object left justified in a sequence of length \textit{width}. Padding is done using the specified \textit{fillbyte} (default is an ASCII space). For \texttt{bytes} objects, the original sequence is returned if \textit{width} is less than or equal to \texttt{len(s)}.

\textbf{Note:} The bytearray version of this method does \textit{not} operate in place - it always produces a new object, even if no changes were made.

bytes.\texttt{lstrip}([chars])
bytearray.\texttt{lstrip}([chars])

Return a copy of the sequence with specified leading bytes removed. The \textit{chars} argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or \texttt{None}, the \textit{chars} argument defaults to removing ASCII whitespace. The \textit{chars} argument is not a prefix; rather, all combinations of its values are stripped:

\begin{verbatim}
>>> b' spacious '.lstrip()
'b spacious'
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
\end{verbatim}

The binary sequence of byte values to remove may be any \texttt{bytes}-like object.

\textbf{Note:} The bytearray version of this method does \textit{not} operate in place - it always produces a new object, even if no changes were made.
bytes.\texttt{rjust}(width[, fillbyte])

Return a copy of the object right justified in a sequence of length \textit{width}. Padding is done using the specified \textit{fillbyte} (default is an ASCII space). For \texttt{bytes} objects, the original sequence is returned if \textit{width} is less than or equal to \texttt{len(s)}.

\textbf{Note:} The bytarray version of this method does \textit{not} operate in place - it always produces a new object, even if no changes were made.

bytes.\texttt{rsplit}(sep=None, maxsplit=-1)

Split the binary sequence into subsequences of the same type, using \texttt{sep} as the delimiter string. If \textit{maxsplit} is given, at most \textit{maxsplit} splits are done, the \textit{rightmost} ones. If \texttt{sep} is not specified or \texttt{None}, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, \texttt{rsplit()} behaves like \texttt{split()} which is described in detail below.

bytes.\texttt{rstrip}([chars])

Return a copy of the sequence with specified trailing bytes removed. The \texttt{chars} argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or \texttt{None}, the \texttt{chars} argument defaults to removing ASCII whitespace. The \texttt{chars} argument is not a suffix; rather, all combinations of its values
are stripped:

```python
>>> b' spacious '.rstrip()
'b' spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

The binary sequence of byte values to remove may be any bytes-like object.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```python
bytes.split(sep=None, maxsplit=-1)
bytearray.split(sep=None, maxsplit=-1)
```

Split the binary sequence into subsequences of the same type, using `sep` as the delimiter string. If `maxsplit` is given and non-negative, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or is `-1`, then there is no limit on the number of splits (all possible splits are made).

If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b','`) returns `[b'1', b'', b'2']`). The `sep` argument may consist of a multibyte sequence (for example, `b'1<>2<>3'.split(b'<>')` returns `[b'1', b'2', b'3']`). Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The `sep` argument may be any bytes-like object.

For example:
If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```python
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b' 1 2 3 '.split()
[b'1', b'2', b'3']
```

`bytes.strip([chars])`
`bytestring.strip([chars])`

Return a copy of the sequence with specified leading and trailing bytes removed. The `chars` argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the `chars` argument defaults to removing ASCII whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped:
The binary sequence of byte values to remove may be any bytes-like object.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytearray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

**bytes**.capitalize()
**bytearray**.capitalize()

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

**bytes**.expandtabs(\(\text{tabsize}=8\))
**bytearray**.expandtabs(\(\text{tabsize}=8\))

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current
column and the given tab size. Tab positions occur every tabsize bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (b'\t'), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (b'\n') or carriage return (b'\r'), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

```python
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01  012  0123  01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012  0123  01234'
```

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

**bytes.isalnum()**

**bytearray.isalnum()**

Return true if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. ASCII decimal digits are those byte values in the sequence b'0123456789'.

For example:
bytes.isalnum()
byterange.isalnum()

Return true if all bytes in the sequence are alphanumeric and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.

For example:

>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False

bytes.isalpha()
byterange.isalpha()

Return true if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.

For example:

>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False

bytes.isascii()
byterange.isascii()

Return true if the sequence is empty or all bytes in the sequence are ASCII, false otherwise. ASCII bytes are in the range 0-0x7F.

New in version 3.7.

bytes.isdigit()
byterange.isdigit()

Return true if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, false otherwise. ASCII decimal digits are those byte values in the sequence b'0123456789'.

For example:
bytes. **islower()**
bytearray. **islower()**

Return true if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, false otherwise.

For example:

```python
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes. **isspace()**
bytearray. **isspace()**

Return true if all bytes in the sequence are ASCII whitespace and the sequence is not empty, false otherwise. ASCII whitespace characters are those byte values in the sequence `b'\t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

bytes. **istitle()**
bytearray. **istitle()**

Return true if the sequence is ASCII titlecase and the sequence is
not empty, false otherwise. See `bytes.title()` for more details on the definition of “titlecase”.

For example:

```python
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()`
`bytearray.isupper()`

Return true if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, false otherwise.

For example:

```python
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`
`bytearray.lower()`

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:
Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
>>> b'Hello World'.lower()
'b'hello world'
```

```
Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
>>> b'Hello World'.lower()
'b'hello world'
```

```
Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
>>> b'Hello World'.lower()
'b'hello world'
```

```
```
bytes.swapcase()
bytestring.swapcase()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```python
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike `str.swapcase()`, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

**Note:** The bytestring version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.title()
bytestring.title()

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining
characters are lowercase. Uncased byte values are left unmodified.

For example:

```python
>>> b'Hello world'.title()
b'Hello World'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```python
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```python
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1] + mo.group(0)[1:],
...                   s)
... >>> titlecase(b"they're bill's friends."
'b"They're Bill's Friends."
```
**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.upper()`

`bytearray.upper()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```python
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

**Note:** The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length `width`. A leading sign prefix (`b'+'/` `b'-'/`) is handled by inserting the padding *after* the sign character rather than before. For `bytes` objects, the original sequence is
returned if \textit{width} is less than or equal to \texttt{len(seq)}.

For example:

\begin{verbatim}
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
\end{verbatim}

\textbf{Note}: The bytearray version of this method does \textit{not} operate in place - it always produces a new object, even if no changes were made.

\textbf{printf-style Bytes Formatting}

\textbf{Note}: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (\texttt{bytes}/\texttt{bytearray}) have one unique built-in operation: the \% operator (modulo). This is also known as the bytes \textit{formatting} or \textit{interpolation} operator. Given \texttt{format \% values} (where \texttt{format} is a bytes object), \% conversion specifications in \texttt{format} are replaced with zero or more elements of \texttt{values}. The effect is similar to using the \texttt{printf()} in the C language.

If \texttt{format} requires a single argument, \texttt{values} may be a single non-tuple object. [5] Otherwise, \texttt{values} must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).
A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `%` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, (somename)).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `*` (asterisk), the actual width is read from the next element of the tuple in `values`, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `.` (dot) followed by the precision. If specified as `*` (an asterisk), the actual precision is read from the next element of the tuple in `values`, and the value to convert comes after the precision.
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object must include a parenthesised mapping key into that dictionary inserted immediately after the `%` character. The mapping key selects the value to be formatted from the mapping. For example:

```python
>>> print(b'%(language)s has %(number)03d quote types.
...     {b'language': b'Python', b'number': 2})
b'Python has 002 quote types.'
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:
<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'#'</td>
<td>The value conversion will use the “alternate form” (where defined below).</td>
</tr>
<tr>
<td>'0'</td>
<td>The conversion will be zero padded for numeric values.</td>
</tr>
<tr>
<td>'-'</td>
<td>The converted value is left adjusted (overrides the '0' conversion if both are given).</td>
</tr>
<tr>
<td>(a space)</td>
<td>A blank should be left before a positive number (or empty string) produced by a signed conversion.</td>
</tr>
<tr>
<td>'+'</td>
<td>A sign character ('+') will precede the conversion (overrides a “space” flag).</td>
</tr>
</tbody>
</table>

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. `%1d` is identical to `%d`.

The conversion types are:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>'i'</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>'o'</td>
<td>Signed octal value.</td>
</tr>
<tr>
<td>'u'</td>
<td>Obsolete type – it is identical to 'd'.</td>
</tr>
<tr>
<td>'x'</td>
<td>Signed hexadecimal (lowercase).</td>
</tr>
<tr>
<td>'X'</td>
<td>Signed hexadecimal (uppercase).</td>
</tr>
<tr>
<td>'e'</td>
<td>Floating point exponential format (lowercase).</td>
</tr>
<tr>
<td>'E'</td>
<td>Floating point exponential format (uppercase).</td>
</tr>
<tr>
<td>'f'</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>'F'</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>'g'</td>
<td>Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
</tr>
<tr>
<td>'G'</td>
<td>Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
</tr>
<tr>
<td>'c'</td>
<td>Single byte (accepts integer or single byte objects).</td>
</tr>
<tr>
<td>'b'</td>
<td>Bytes (any object that follows the buffer protocol or has <strong>bytes</strong>()).</td>
</tr>
<tr>
<td>'s'</td>
<td>'s' is an alias for 'b' and should only be used for Python2/3 code bases.</td>
</tr>
<tr>
<td>'a'</td>
<td>Bytes (converts any Python object using repr(obj).encode('ascii','backslashreplace)).</td>
</tr>
<tr>
<td>'r'</td>
<td>'r' is an alias for 'a' and should only be used for Python2/3 code bases.</td>
</tr>
<tr>
<td>'%'</td>
<td>No argument is converted, results in a '%' character in the result.</td>
</tr>
</tbody>
</table>

Notes:

1. The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.

2. The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.

3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.
4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. If precision is \(N\), the output is truncated to \(N\) characters.

6. \texttt{b'\%s'} is deprecated, but will not be removed during the 3.x series.

7. \texttt{b'\%r'} is deprecated, but will not be removed during the 3.x series.

8. See PEP 237.

\begin{tcolorbox}
\textbf{Note:} The bytearray version of this method does not operate in place - it always produces a new object, even if no changes were made.
\end{tcolorbox}

\textbf{See also:} PEP 461 - Adding \% formatting to bytes and bytearray

\textit{New in version 3.5.}

\section*{Memory Views}

\texttt{memoryview} objects allow Python code to access the internal data of an object that supports the \texttt{buffer protocol} without copying.

\begin{tcolorbox}
\texttt{class memoryview(obj)}

Create a \texttt{memoryview} that references \texttt{obj}. \texttt{obj} must support the buffer protocol. Built-in objects that support the buffer protocol
\end{tcolorbox}
include bytes and bytearray.

A memoryview has the notion of an element, which is the atomic memory unit handled by the originating object obj. For many simple types such as bytes and bytearray, an element is a single byte, but other types such as array.array may have bigger elements.

len(view) is equal to the length of tolist. If view.ndim = 0, the length is 1. If view.ndim = 1, the length is equal to the number of elements in the view. For higher dimensions, the length is equal to the length of the nested list representation of the view. The itemsize attribute will give you the number of bytes in a single element.

A memoryview supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```python
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If format is one of the native format specifiers from the struct module, indexing with an integer or a tuple of integers is also supported and returns a single element with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed
with tuples of exactly \( ndim \) integers where \( ndim \) is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```python
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -3
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```python
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rva
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z123fg')
```
One-dimensional memoryviews of hashable (read-only) types with formats ‘B’, ‘b’ or ‘c’ are also hashable. The hash is defined as
\[
\text{hash}(m) == \text{hash}(m\text{.tobytes()})
\]

```python
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::2]) == hash(b'abcefg'[::2])
True
```

*Changed in version 3.3:* One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats ‘B’, ‘b’ or ‘c’ are now hashable.

*Changed in version 3.4:* memoryview is now registered automatically with `collections.abc.Sequence`

*Changed in version 3.5:* memoryviews can now be indexed with tuple of integers.

`memoryview` has several methods:

```python
___eq___(exporter)
```

A memoryview and a [PEP 3118](https://www.python.org/dev/peps/pep-0318/) exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands’ respective format codes are interpreted using `struct` syntax.

For the subset of `struct` format strings currently supported by
tolist(), v and w are equal if v.tolist() == w.tolist:

```python
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[:::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the `struct` module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```python
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = ["x", c_long], ("y", c_long)
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```
Note that, as with floating point numbers, \( v \) is \( w \) does not imply \( v == w \) for memoryview objects.

*Changed in version 3.3:* Previous versions compared the raw memory disregarding the item format and the logical array structure.

**tobytes()**

Return the data in the buffer as a bytestring. This is equivalent to calling the `bytes` constructor on the memoryview.

```python
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. `tobytes()` supports all format strings, including those that are not in `struct` module syntax.

**hex()**

Return a string object containing two hexadecimal digits for each byte in the buffer.

```python
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

*New in version 3.5.*

**tolist()**

Return the data in the buffer as a list of elements.
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]

*Changed in version 3.3:* `tolist()` now supports all single character native formats in `struct` module syntax as well as multi-dimensional representations.

**release()**

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a `bytearray` would temporarily forbid resizing); therefore, calling `release()` is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a `ValueError` (except `release()` itself which can be called multiple times):

```python
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memory
```

The context management protocol can be used for a similar effect, using the `with` statement:
```python
>>> with memoryview(b'abc') as m:
...     m[0]
...     97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

*New in version 3.2.*

**cast**(*format*, *shape*)

Cast a memoryview to a new format or shape. *shape* defaults to [byte_length//new_itemsize], which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

The destination format is restricted to a single element native format in struct syntax. One of the formats must be a byte format (‘B’, ‘b’ or ‘c’). The byte length of the result must be the same as the original length.

Cast 1D/long to 1D/unsigned bytes:

```python
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
```
Cast 1D/unsigned bytes to 1D/char:

```python
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format 'B'

```python
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/int64 to 1D/signed char:

```python
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]
>>> y.format
'i'
>>> y.itemsize
```

```python
24
```
Cast 1D/unsigned char to 2D/unsigned long:

```python
>>> buf = struct.pack("L"*6, *list(range(6))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

*New in version 3.3.*

*Changed in version 3.5:* The source format is no longer restricted when casting to a byte view.

There are also several readonly attributes available:

```python
obj
```
The underlying object of the memoryview:

```python
>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

*New in version 3.3.*

**nbytes**

```
nbytes == product(shape) * itemsize == len(m.tobytes()). This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to len(m):
```

```python
>>> import array
>>> a = array.array('i', [1, 2, 3, 4, 5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```python
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in x = memoryview(buf)
```
>>> y = x.cast('d', shape=[3, 4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96

New in version 3.3.

readonly
A bool indicating whether the memory is read only.

format
A string containing the format (in `struct` module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. `tolist()`) are restricted to native single element formats.

Changed in version 3.3: format `B` is now handled according to the `struct` module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize
The size in bytes of each element of the memoryview:

```python
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```
**ndim**
An integer indicating how many dimensions of a multi-dimensional array the memory represents.

**shape**
A tuple of integers the length of `ndim` giving the shape of the memory as an N-dimensional array.

*Changed in version 3.3:* An empty tuple instead of `None` when `ndim = 0`.

**strides**
A tuple of integers the length of `ndim` giving the size in bytes to access each element for each dimension of the array.

*Changed in version 3.3:* An empty tuple instead of `None` when `ndim = 0`.

**suboffsets**
Used internally for PIL-style arrays. The value is informational only.

**c_contiguous**
A bool indicating whether the memory is C-contiguous.

*New in version 3.3.*

**f_contiguous**
A bool indicating whether the memory is Fortran contiguous.

*New in version 3.3.*

**contiguous**
A bool indicating whether the memory is **contiguous**.

*New in version 3.3.*
Set Types — \texttt{set, frozenset}

A \texttt{set} object is an unordered collection of distinct \texttt{hashable} objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in \texttt{dict, list,} and \texttt{tuple} classes, and the \texttt{collections} module.)

Like other collections, sets support \texttt{x in set, len(set)}, and \texttt{for x in set}. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, \texttt{set} and \texttt{frozenset}. The \texttt{set} type is mutable — the contents can be changed using methods like \texttt{add()} and \texttt{remove()}. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The \texttt{frozenset} type is immutable and \texttt{hashable} — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: \{'jack', 'sjoerd'\}, in addition to the \texttt{set} constructor.

The constructors for both classes work the same:

\begin{verbatim}
class set([iterable])
class frozenset([iterable])
\end{verbatim}

Return a new \texttt{set} or \texttt{frozenset} object whose elements are taken
from *iterable*. The elements of a set must be *hashable*. To represent sets of sets, the inner sets must be *frozenset* objects. If *iterable* is not specified, a new empty set is returned.

Instances of *set* and *frozenset* provide the following operations:

### `len(s)`
Return the number of elements in set *s* (cardinality of *s*).

### `x in s`
Test *x* for membership in *s*.

### `x not in s`
Test *x* for non-membership in *s*.

### `isdisjoint(other)`
Return *True* if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

### `issubset(other)`
*set* <= *other*
Test whether every element in the set is in *other*.

### `set < other`
Test whether the set is a proper subset of *other*, that is, *set* <= *other* and *set* != *other*.

### `issuperset(other)`
*set* >= *other*
Test whether every element in *other* is in the set.

### `set > other`
Test whether the set is a proper superset of other, that is, set >= other and set != other.

union(*others)
set | other | ...
Return a new set with elements from the set and all others.

intersection(*others)
set & other & ...
Return a new set with elements common to the set and all others.

difference(*others)
set - other - ...
Return a new set with elements in the set that are not in the others.

symmetric_difference(other)
set ^ other
Return a new set with elements in either the set or other but not both.

copy()
Return a shallow copy of the set.

Note, the non-operator versions of union(), intersection(), difference(), and symmetric_difference(), issubset(), and issuperset() methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like set('abc') & 'cbs' in favor of the more readable set('abc').intersection('cbs').
Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')]).`

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so all of the following return `False`: `a<b`, `a==b`, or `a>b`.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements, like dictionary keys, must be `hashable`.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

```
update(*others)
set  |=  other  |  ...
    Update the set, adding elements from all others.
```
**intersection_update**(*others*)
```
set &= other & ...
```
Update the set, keeping only elements found in it and all others.

**difference_update**(*others*)
```
set -= other | ...
```
Update the set, removing elements found in others.

**symmetric_difference_update**(other)
```
set ^= other
```
Update the set, keeping only elements found in either set, but not in both.

**add**(elem)
```
Add element elem to the set.
```

**remove**(elem)
```
Remove element elem from the set. Raises KeyError if elem is not contained in the set.
```

**discard**(elem)
```
Remove element elem from the set if it is present.
```

**pop**()
```
Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.
```

**clear**()
```
Remove all elements from the set.
```

Note, the non-operator versions of the update(), intersection_update(), difference_update(), and symmetric_difference_update() methods will accept any
iterable as an argument.

Note, the *elem* argument to the *__contains__()*, *remove()* , and *discard()* methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*. 
A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (For other containers see the built-in list, set, and tuple classes, and the collections module.)

A dictionary’s keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}, or by the dict constructor.

class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping
object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to 
{"one": 1, "two": 2, "three": 3}:

```python
given:
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

**len(d)**

Return the number of items in the dictionary d.
d[key]

Return the item of d with key key. Raises a KeyError if key is not in the map.

If a subclass of dict defines a method __missing__() and key is not present, the d[key] operation calls that method with the key key as argument. The d[key] operation then returns or raises whatever is returned or raised by the __missing__(key) call. No other operations or methods invoke __missing__(). If __missing__() is not defined, KeyError is raised. __missing__() must be a method; it cannot be an instance variable.

```python
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
... >>> c = Counter()
... >>> c['red']
0
... >>> c['red'] += 1
... >>> c['red']
1
```

The example above shows part of the implementation of collections.Counter. A different __missing__ method is used by collections.defaultdict.

d[key] = value

Set d[key] to value.

del d[key]

Remove d[key] from d. Raises a KeyError if key is not in the map.
key in d
   Return True if d has a key key, else False.

key not in d
   Equivalent to not key in d.

iter(d)
   Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).

clear()
   Remove all items from the dictionary.

copy()
   Return a shallow copy of the dictionary.

classmethod fromkeys(iterable[, value])
   Create a new dictionary with keys from iterable and values set to value.

   fromkeys() is a class method that returns a new dictionary. value defaults to None.

get(key[, default])
   Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items()
   Return a new view of the dictionary's items ((key, value) pairs). See the documentation of view objects.

keys()
Return a new view of the dictionary's keys. See the documentation of view objects.

\[ \textbf{pop}(key[, \text{default}]) \]
If \textit{key} is in the dictionary, remove it and return its value, else return \textit{default}. If \textit{default} is not given and \textit{key} is not in the dictionary, a \texttt{KeyError} is raised.

\[ \textbf{popitem}() \]
Remove and return a \((key, value)\) pair from the dictionary. Pairs are returned in \texttt{LIFO} order.

\[ \text{popitem}() \] is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling \text{popitem}() raises a \texttt{KeyError}.

\textit{Changed in version 3.7:} LIFO order is now guaranteed. In prior versions, \text{popitem}() would return an arbitrary key/value pair.

\[ \textbf{setdefault}(key[, \text{default}]) \]
If \textit{key} is in the dictionary, return its value. If not, insert \textit{key} with a value of \textit{default} and return \textit{default}. \textit{default} defaults to \texttt{None}.

\[ \textbf{update}([other]) \]
Update the dictionary with the key/value pairs from \textit{other}, overwriting existing keys. Return \texttt{None}.

\textit{update}() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: \texttt{d.update(red=1, blue=2)}.
**values()**

Return a new view of the dictionary's values. See the documentation of view objects.

Dictionaries compare equal if and only if they have the same (key, value) pairs. Order comparisons ('<', '<=', '>', '==', '>') raise TypeError.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```python
>>> d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d['one'] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d['two']
>>> d['two'] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

*Changed in version 3.7:* Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

**See also:** types.MappingProxyType can be used to create a read-only view of a dict.
Dictionary view objects

The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are view objects. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

**len(dictview)**

Return the number of entries in the dictionary.

**iter(dictview)**

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

*Changed in version 3.7: Dictionary order is guaranteed to be insertion order.*

**x in dictview**

Return `True` if `x` is in the underlying dictionary's keys, values or items (in the latter case, `x` should be a `(key, value)` tuple).

Keys views are set-like since their entries are unique and hashable. If
all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are available (for example, `==`, `<`, or `^`).

An example of dictionary view usage:

```python
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```
Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

```python
contextmanager.__enter__()

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the as clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

```python
contextmanager.__exit__(exc_type, exc_val, exc_tb)

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are None.
Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the `contextlib` module for some examples.

Python’s `generators` and the `contextlib.contextmanager` decorator provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextmanager` decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal
Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.
Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

Modules

The only special operation on a module is attribute access: \texttt{m.name}, where \texttt{m} is a module and \texttt{name} accesses a name defined in \texttt{m}'s symbol table. Module attributes can be assigned to. (Note that the \texttt{import} statement is not, strictly speaking, an operation on a module object; \texttt{import foo} does not require a module object named \texttt{foo} to exist, rather it requires an (external) \texttt{definition} for a module named \texttt{foo} somewhere.)

A special attribute of every module is \texttt{__dict__}. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the \texttt{__dict__} attribute is not possible (you can write \texttt{m.__dict__['a'] = 1}, which defines \texttt{m.a} to be \texttt{1}, but you can't write \texttt{m.__dict__ = {}}). Modifying \texttt{__dict__} directly is not recommended.

Modules built into the interpreter are written like this: \texttt{<module 'sys' (built-in)>}. If loaded from a file, they are written as \texttt{<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>}.

Classes and Class Instances

See \texttt{Objects, values and types} and \texttt{Class definitions} for these.
Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See Function definitions for more information.

Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a bound method (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the
underlying function object (meth.__func__), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an AttributeError being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
>>> c.method.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See The standard type hierarchy for more information.

### Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in compile() function and can be extracted from function objects through their __code__ attribute. See also the code module.

A code object can be executed or evaluated by passing it (instead of a source string) to the exec() or eval() built-in functions.
See The standard type hierarchy for more information.

**Type Objects**

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

**The Null Object**

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

It is written as `None`.

**The Ellipsis Object**

This object is commonly used by slicing (see Slicings). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)()` produces the `Ellipsis` singleton.

It is written as `Ellipsis` or `...`.

**The NotImplemented Object**
This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See Comparisons for more information. There is exactly one NotImplemented object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

### Boolean Values

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function `bool()` can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

They are written as `False` and `True`, respectively.

### Internal Objects

See The standard type hierarchy for this information. It describes stack frame objects, traceback objects, and slice objects.
Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

`object.__dict__`
A dictionary or other mapping object used to store an object's (writable) attributes.

`instance.__class__`
The class to which a class instance belongs.

`class.__bases__`
The tuple of base classes of a class object.

`definition.__name__`
The name of the class, function, method, descriptor, or generator instance.

`definition.__qualname__`
The qualified name of the class, function, method, descriptor, or generator instance.

*New in version 3.3.*

`class.__mro__`
This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

`class.mro()`
This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class
instantiation, and its result is stored in __mro__.

class.__subclasses__()  
Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```python
>>> int.__subclasses__()
<class 'bool'>
```

Footnotes

[1] Additional information on these special methods may be found in the Python Reference Manual (Basic customization).

[2] As a consequence, the list [1, 2] is considered equal to [1.0, 2.0], and similarly for tuples.

[3] They must have since the parser can’t tell the type of the operands.

(1, 2, 3, 4) Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

[4] (1, 2) To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.
Built-in Exceptions

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class’s constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under User-defined Exceptions.

When raising (or re-raising) an exception in an `except` or `finally` clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually
displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is false.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.
Base classes

The following exceptions are used mostly as base classes for other exceptions.

**exception BaseException**

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

**args**

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `OSError`) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

**with_traceback(tb)**

This method sets `tb` as the new traceback for the exception and returns the exception object. It is usually used in exception handling code like this:

```python
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

**exception Exception**

All built-in, non-system-exiting exceptions are derived from this...
class. All user-defined exceptions should also be derived from this class.

(exception **ArithmeticError**)
The base class for those built-in exceptions that are raised for various arithmetic errors: **OverflowError**, **ZeroDivisionError**, **FloatingPointError**.

(exception **BufferError**)
Raised when a buffer related operation cannot be performed.

(exception **LookupError**)
The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: **IndexError**, **KeyError**. This can be raised directly by `codecs.lookup()`.
Concrete exceptions

The following exceptions are the exceptions that are usually raised.

exception **AssertionError**
   Raised when an `assert` statement fails.

exception **AttributeError**
   Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, **TypeError** is raised.)

exception **EOFError**
   Raised when the `input()` function hits an end-of-file condition (EOF) without reading any data. (N.B.: the `io.IOBase.read()` and `io.IOBase.readline()` methods return an empty string when they hit EOF.)

exception **FloatingPointError**
   Not currently used.

exception **GeneratorExit**
   Raised when a generator or coroutine is closed; see `generator.close()` and `coroutine.close()`. It directly inherits from **BaseException** instead of **Exception** since it is technically not an error.

exception **ImportError**
   Raised when the `import` statement has troubles trying to load a module. Also raised when the “from list” in `from ... import` has a name that cannot be found.
The name and path attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively.

Changed in version 3.3: Added the name and path attributes.

exception **ModuleNameNotFoundError**

A subclass of `ImportError` which is raised by `import` when a module could not be located. It is also raised when None is found in `sys.modules`.

New in version 3.6.

exception **IndexError**

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, `TypeError` is raised.)

exception **KeyError**

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception **KeyboardInterrupt**

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. The exception inherits from `BaseException` so as to not be accidentally caught by code that catches `Exception` and thus prevent the interpreter from exiting.

exception **MemoryError**

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated
value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

**exception NameError**

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

**exception NotImplementedError**

This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

**Note:** It should not be used to indicate that an operator or method is not meant to be supported at all – in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

**Note:** `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See `NotImplemented` for details on when to use it.

**exception OSError([arg])**

**exception OSError(errno, strerror[, filename[, winerror[, filename2]]]])**
This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to None if not specified. For backwards compatibility, if three arguments are passed, the args attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of OSError, as described in OS exceptions below. The particular subclass depends on the final errno value. This behaviour only occurs when constructing OSError directly or via an alias, and is not inherited when subclassing.

**errno**

A numeric error code from the C variable errno.

**winerror**

Under Windows, this gives you the native Windows error code. The errno attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the winerror constructor argument is an integer, the errno attribute is determined from the Windows error code, and the errno argument is ignored. On other platforms, the winerror argument is ignored, and the winerror attribute does not exist.

**strerror**

The corresponding error message, as provided by the operating
system. It is formatted by the C functions perror() under POSIX, and FormatMessage() under Windows.

filename filename2

For exceptions that involve a file system path (such as open() or os.unlink()), filename is the file name passed to the function. For functions that involve two file system paths (such as os.rename()), filename2 corresponds to the second file name passed to the function.

*Changed in version 3.3:* EnvironmentError, IOError, WindowsError, socket.error, select.error and mmap.error have been merged into OSError, and the constructor may return a subclass.

*Changed in version 3.4:* The filename attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the filesystem encoding. Also, the filename2 constructor argument and attribute was added.

*exception OverflowError*

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise MemoryError than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating point exception handling in C, most floating point operations are not checked.

*exception RecursionError*

This exception is derived from RuntimeError. It is raised when
the interpreter detects that the maximum recursion depth (see \texttt{sys.getrecursionlimit()}) is exceeded.

\textit{New in version 3.5:} Previously, a plain \texttt{RuntimeError} was raised.

\texttt{exception ReferenceError}

This exception is raised when a weak reference proxy, created by the \texttt{weakref.proxy()} function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the \texttt{weakref} module.

\texttt{exception RuntimeError}

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

\texttt{exception StopIteration}

Raised by built-in function \texttt{next()} and an iterator’s \texttt{__next__()} method to signal that there are no further items produced by the iterator.

The exception object has a single attribute \texttt{value}, which is given as an argument when constructing the exception, and defaults to \texttt{None}.

When a \texttt{generator} or \texttt{coroutine} function returns, a new \texttt{StopIteration} instance is raised, and the value returned by the function is used as the value parameter to the constructor of the exception.

If a generator code directly or indirectly raises \texttt{StopIteration}, it is converted into a \texttt{RuntimeError} (retaining the \texttt{StopIteration} as the new exception’s cause).
**Changed in version 3.3:** Added **value** attribute and the ability for generator functions to use it to return a value.

**Changed in version 3.5:** Introduced the **RuntimeError** transformation via `from __future__ import generator_stop`, see [PEP 479](https://www.python.org/dev/peps/pep-0479/).

**Changed in version 3.7:** Enable [PEP 479](https://www.python.org/dev/peps/pep-0479/) for all code by default: a **StopIteration** error raised in a generator is transformed into a **RuntimeError**.

**exception** `StopAsyncIteration`
Must be raised by `__anext__()` method of an asynchronous iterator object to stop the iteration.

**New in version 3.5.**

**exception** `SyntaxError`
Raised when the parser encounters a syntax error. This may occur in an **import** statement, in a call to the built-in functions `exec()` or `eval()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

**exception** `IndentationError`
Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

**exception** `TabError`
Raised when indentation contains an inconsistent use of tabs and
spaces. This is a subclass of IndentationError.

**exception SystemError**

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (sys.version; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

**exception SystemExit**

This exception is raised by the sys.exit() function. It inherits from BaseException instead of Exception so that it is not accidentally caught by code that catches Exception. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to sys.exit(). If the value is an integer, it specifies the system exit status (passed to C's exit() function); if it is None, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to sys.exit() is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The os._exit() function can be used if it is absolutely positively necessary to exit immediately
(for example, in the child process after a call to `os.fork()`).

**code**

The exit status or error message that is passed to the constructor. (Defaults to None.)

**exception** `TypeError`

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, `NotImplementedError` is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a `list` when an `int` is expected) should result in a `TypeError`, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a `ValueError`.

**exception** `UnboundLocalError`

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

**exception** `UnicodeError`

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

`UnicodeError` has attributes that describe the encoding or decoding error. For example,
err.object[err.start:err.end] gives the particular invalid input that the codec failed on.

**encoding**
The name of the encoding that raised the error.

**reason**
A string describing the specific codec error.

**object**
The object the codec was attempting to encode or decode.

**start**
The first index of invalid data in **object**.

**end**
The index after the last invalid data in **object**.

*exception* **UnicodeEncodeError**
Raised when a Unicode-related error occurs during encoding. It is a subclass of **UnicodeError**.

*exception* **UnicodeDecodeError**
Raised when a Unicode-related error occurs during decoding. It is a subclass of **UnicodeError**.

*exception* **UnicodeTranslateError**
Raised when a Unicode-related error occurs during translating. It is a subclass of **UnicodeError**.

*exception* **ValueError**
Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is
not described by a more precise exception such as `IndexError`.

**exception ZeroDivisionError**

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of `OSError`.

**exception EnvironmentError**

**exception IOError**

**exception WindowsError**

Only available on Windows.

**OS exceptions**

The following exceptions are subclasses of `OSError`, they get raised depending on the system error code.

**exception BlockingIOError**

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno` EAGAIN, EALREADY, EWOULDBLOCK and EINPROGRESS.

In addition to those of `OSError`, `BlockingIOError` can have one more attribute:

**characters_written**

An integer containing the number of characters written to the stream before it blocked. This attribute is available when using
the buffered I/O classes from the `io` module.

**exception ChildProcessError**
Raised when an operation on a child process failed. Corresponds to `errno` ECHILD.

**exception ConnectionError**
A base class for connection-related issues.

Subclasses are `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` and `ConnectionResetError`.

**exception BrokenPipeError**
A subclass of `ConnectionError`, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno` EPIPE and ESHUTDOWN.

**exception ConnectionAbortedError**
A subclass of `ConnectionError`, raised when a connection attempt is aborted by the peer. Corresponds to `errno` ECONNABORTED.

**exception ConnectionRefusedError**
A subclass of `ConnectionError`, raised when a connection attempt is refused by the peer. Corresponds to `errno` ECONNREFUSED.

**exception ConnectionResetError**
A subclass of `ConnectionError`, raised when a connection is reset by the peer. Corresponds to `errno` ECONNRESET.
exception **FileExistsError**
Raised when trying to create a file or directory which already exists. Corresponds to `errno EEXIST`.

exception **FileNotFoundError**
Raised when a file or directory is requested but doesn’t exist. Corresponds to `errno ENOENT`.

exception **InterruptedError**
Raised when a system call is interrupted by an incoming signal. Corresponds to `errno EINTR`.

*Changed in version 3.5:* Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](https://www.python.org/dev/peps/pep-0475/) for the rationale), instead of raising `InterruptedError`.

exception **IsADirectoryError**
Raised when a file operation (such as `os.remove()`) is requested on a directory. Corresponds to `errno EISDIR`.

exception **NotADirectoryError**
Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. Corresponds to `errno ENOTDIR`.

exception **PermissionError**
Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to `errno EACCES` and `EPERM`.

exception **ProcessLookupError**
Raised when a given process doesn’t exist. Corresponds to \texttt{errno ESRCH}.

\textit{exception TimeoutError}

Raised when a system function timed out at the system level. Corresponds to \texttt{errno ETIMEDOUT}.

\textit{New in version 3.3:} All the above \texttt{OSError} subclasses were added.

\textbf{See also:} \texttt{PEP 3151} - Reworking the OS and IO exception hierarchy
Warnings

The following exceptions are used as warning categories; see the Warning Categories documentation for more details.

exception **Warning**
Base class for warning categories.

exception **UserWarning**
Base class for warnings generated by user code.

exception **DeprecationWarning**
Base class for warnings about deprecated features when those warnings are intended for other Python developers.

exception **PendingDeprecationWarning**
Base class for warnings about features which will be deprecated in the future.

exception **SyntaxWarning**
Base class for warnings about dubious syntax.

exception **RuntimeWarning**
Base class for warnings about dubious runtime behavior.

exception **FutureWarning**
Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

exception **ImportWarning**
Base class for warnings about probable mistakes in module imports.
exception **UnicodeWarning**
   Base class for warnings related to Unicode.

exception **BytesWarning**
   Base class for warnings related to bytes and bytearray.

exception **ResourceWarning**
   Base class for warnings related to resource usage. Ignored by the default warning filters.

*New in version 3.2.*
Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
    +-- SystemExit
    +-- KeyboardInterrupt
    +-- GeneratorExit
    +-- Exception
        +-- StopIteration
        +-- StopAsyncIteration
        +-- ArithmeticError
            |   +-- FloatingPointError
            |     +-- OverflowError
            |     +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- BufferError
        +-- EOFError
        +-- ImportError
            |   +-- ModuleNotFoundError
        +-- LookupError
            |   +-- IndexError
            |     +-- KeyError
        +-- MemoryError
        +-- NameError
            |   +-- UnboundLocalError
        +-- OSError
            |   +-- BlockingIOError
            |     +-- ChildProcessError
            |          +-- ConnectionError
            |          |   +-- BrokenPipeError
            |          |     +-- ConnectionAbortedError
            |          |     +-- ConnectionRefusedError
            |          |     +-- ConnectionResetError
```
FileExistsError
|    | FileNotFoundError
|    | InterruptedError
|    | IsADirectoryError
|    | NotADirectoryError
|    | PermissionError
|    | ProcessLookupError
|    | TimeoutError
|    | ReferenceError
|    | RuntimeError
|    | NotImplementedError
|    | RecursionError
|    | SyntaxError
|    | IndentationError
|    | TabError
|    | SystemError
|    | TypeError
|    | ValueError
|    | UnicodeError
|    | UnicodeDecodeError
|    | UnicodeEncodeError
|    | UnicodeTranslateError
|    | Warning
|    | DeprecationWarning
|    | PendingDeprecationWarning
|    | RuntimeWarning
|    | SyntaxWarning
|    | UserWarning
|    | FutureWarning
|    | ImportWarning
|    | UnicodeWarning
|    | BytesWarning
|    | ResourceWarning
Text Processing Services

The modules described in this chapter provide a wide range of string manipulation operations and other text processing services.

The codecs module described under Binary Data Services is also highly relevant to text processing. In addition, see the documentation for Python’s built-in string type in Text Sequence Type — str.

- **string** — Common string operations
  - String constants
  - Custom String Formatting
  - Format String Syntax
    - Format Specification Mini-Language
    - Format examples
  - Template strings
  - Helper functions
- **re** — Regular expression operations
  - Regular Expression Syntax
  - Module Contents
  - Regular Expression Objects
  - Match Objects
  - Regular Expression Examples
    - Checking for a Pair
    - Simulating scanf()
    - search() vs. match()
    - Making a Phonebook
    - Text Munging
    - Finding all Adverbs
    - Finding all Adverbs and their Positions
    - Raw String Notation
    - Writing a Tokenizer
• difflib — Helpers for computing deltas
  • SequenceMatcher Objects
  • SequenceMatcher Examples
  • Differ Objects
  • Differ Example
  • A command-line interface to difflib
• textwrap — Text wrapping and filling
• unicodedata — Unicode Database
• stringprep — Internet String Preparation
• readline — GNU readline interface
  • Init file
  • Line buffer
  • History file
  • History list
  • Startup hooks
  • Completion
  • Example
• rlcompleter — Completion function for GNU readline
  • Completer Objects
string — Common string operations

Source code: Lib/string.py

See also: Text Sequence Type — str

String Methods
String constants

The constants defined in this module are:

string. **ascii_letters**
   The concatenation of the **ascii_lowercase** and **ascii_uppercase** constants described below. This value is not locale-dependent.

string. **ascii_lowercase**
   The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.

string. **ascii_uppercase**
   The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.

string. **digits**
   The string '0123456789'.

string. **hexdigits**
   The string '0123456789abcdefABCDEF'.

string. **octdigits**
   The string '01234567'.

string. **punctuation**
   String of ASCII characters which are considered punctuation characters in the C locale.

string. **printable**
   String of ASCII characters which are considered printable. This is a
combination of digits, ascii_letters, punctuation, and whitespace.

string. whitespace
A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.
Custom String Formatting

The built-in string class provides the ability to do complex variable substitutions and value formatting via the `format()` method described in PEP 3101. The `Formatter` class in the `string` module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in `format()` method.

```python
class string.Formatter
```

The `Formatter` class has the following public methods:

```python
format(format_string, *args, **kwargs)
```

The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls `vformat()`.

*Changed in version 3.7:* A format string argument is now positional-only.

```python
vformat(format_string, args, kwargs)
```

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the `*args` and `**kwargs` syntax. `vformat()` does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the `Formatter` defines a number of methods that are
intended to be replaced by subclasses:

**parse**(*format_string*)
Loop over the format_string and return an iterable of tuples (*literal_text*, *field_name*, *format_spec*, *conversion*). This is used by **vformat()** to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be None.

**get_field**(*field_name*, *args*, *kwargs*)
Given *field_name* as returned by **parse()** (see above), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form defined in PEP 3101, such as “0[name]” or “label.title”. *args* and *kwargs* are as passed in to **vformat()**. The return value *used_key* has the same meaning as the *key* parameter to **get_value()**.

**get_value**(*key*, *args*, *kwargs*)
Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to **vformat()**, and the *kwargs* parameter is set to the dictionary of keyword arguments.
For compound field names, these functions are only called for the first component of the field name; subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression '0.name' would cause get_value() to be called with a key argument of 0. The name attribute will be looked up after get_value() returns by calling the built-in getattr() function.

If the index or keyword refers to an item that does not exist, then an IndexError or KeyError should be raised.

checkUnusedArgs(used_args, args, kwargs)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the args and kwargs that was passed to vformat. The set of unused args can be calculated from these parameters. checkUnusedArgs() is assumed to raise an exception if the check fails.

formatField(value, format_spec)

formatField() simply calls the global format() built-in. The method is provided so that subclasses can override it.

convertField(value, conversion)

Converts the value (returned by get_field()) given a conversion type (as in the tuple returned by the parse() method). The default version understands ‘s’ (str), ‘r’ (repr) and ‘a’ (ascii) conversion types.
Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax). The syntax is related to that of formatted string literals, but there are differences.

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] "}"
field_name ::= arg_name ("." attribute_name)
arg_name ::= [identifier | digit+]
attribute_name ::= identifier
element_index ::= digit+ | index_string
index_string ::= <any source character except "]">+
conversion ::= "r" | "s" | "a"
format_spec ::= <described in the next section>
```

In less formal terms, the replacement field can start with a `field_name` that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The `field_name` is optionally followed by a `conversion` field, which is preceded by an exclamation point `!!`, and a `format_spec`, which is preceded by a colon `:`. These specify a non-default format for the replacement value.
See also the Format Specification Mini-Language section.

The field_name itself begins with an arg_name that is either a number or a keyword. If it’s a number, it refers to a positional argument, and if it’s a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, … in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, … will be automatically inserted in that order. Because arg_name is not quoted-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings '10' or ':-]') within a format string. The arg_name can be followed by any number of index or attribute expressions. An expression of the form '.name' selects the named attribute using getattr(), while an expression of the form '[index]' does an index lookup using __getitem__().

Changed in version 3.1: The positional argument specifiers can be omitted for str.format(), so '{} {}'.format(a, b) is equivalent to '{0} {1}'.format(a, b).

Changed in version 3.4: The positional argument specifiers can be omitted for Formatter.

Some simple format string examples:

"First, thou shalt count to {0}"  # References first
"Bring me a {}"                  # Implicitly refer
"From {} to {}"                  # Same as "From {0}
"My quest is {name}"            # References keyword
"Weight in tons {0.weight}"     # 'weight' attribute
"Units destroyed: {players[0]}"  # First element of

The conversion field causes a type coercion before formatting. Normally, the job of formatting a value is done by the __format__()
method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

Three conversion flags are currently supported: `!s` which calls `str()` on the value, `!r` which calls `repr()` and `!a` which calls `ascii()`.

Some examples:

```
"Harold's a clever {0!s}"  # Calls str() on the argument first
"Bring out the holy {name!r}"  # Calls repr() on the argument first
"More {!a}"  # Calls ascii() on the argument first
```

The `format_spec` field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the `format_spec`.

Most built-in types support a common formatting mini-language, which is described in the next section.

A `format_spec` field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the `format_spec` are substituted before the `format_spec` string is interpreted. This allows the formatting of a value to be dynamically specified.

See the Format examples section for some examples.

**Format Specification Mini-Language**
“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax and Formatted string literals). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string (""") produces the same result as if you had called `str()` on the value. A non-empty format string typically modifies the result.

The general form of a standard format specifier is:

```plaintext
format_spec ::= [[fill]align][sign][#][0][width][grouping_option][precision][type]
fill ::= <any character>
align ::= "<" | ">" | ";" | "^"
sign ::= "+" | "-" | " "
width ::= digit+
grouping_option ::= ";" | ","
precision ::= digit+
type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" |

```

If a valid `align` value is specified, it can be preceded by a `fill` character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace ("{" or "}") as the `fill` character in a formatted string literal or when using the `str.format()` method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn’t affect the `format()`
function.

The meaning of the various alignment options is as follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'&lt;'</td>
<td>Forces the field to be left-aligned within the available space (this is the default for most objects).</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>Forces the field to be right-aligned within the available space (this is the default for numbers).</td>
</tr>
<tr>
<td>'='</td>
<td>Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.</td>
</tr>
<tr>
<td>'^'</td>
<td>Forces the field to be centered within the available space.</td>
</tr>
</tbody>
</table>

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The sign option is only valid for number types, and can be one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'+'</td>
<td>indicates that a sign should be used for both positive as well as negative numbers.</td>
</tr>
<tr>
<td>'-'</td>
<td>indicates that a sign should be used only for negative numbers (this is the default behavior).</td>
</tr>
</tbody>
</table>

space  indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.
The `#` option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective `'0b'`, `'0o'`, or `'0x'` to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

*Changed in version 3.1:* Added the ',' option (see also PEP 378).

The '_' option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

*Changed in version 3.6:* Added the '_' option (see also PEP 515).

`width` is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the `width` field by a zero (`'0'`) character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of `'0'` with an alignment
The precision is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The precision is not allowed for integer values.

Finally, the type determines how the data should be presented.

The available string presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'s'</td>
<td>String format. This is the default type for strings and may be omitted.</td>
</tr>
<tr>
<td>None</td>
<td>The same as 's'.</td>
</tr>
</tbody>
</table>

The available integer presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>Binary format. Outputs the number in base 2.</td>
</tr>
<tr>
<td>'c'</td>
<td>Character. Converts the integer to the corresponding unicode character before printing.</td>
</tr>
<tr>
<td>'d'</td>
<td>Decimal Integer. Outputs the number in base 10.</td>
</tr>
<tr>
<td>'o'</td>
<td>Octal format. Outputs the number in base 8.</td>
</tr>
<tr>
<td>'x'</td>
<td>Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.</td>
</tr>
<tr>
<td>'X'</td>
<td>Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.</td>
</tr>
</tbody>
</table>
In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'e'</td>
<td>Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6.</td>
</tr>
<tr>
<td>'E'</td>
<td>Exponent notation. Same as 'e' except it uses an upper case ‘E’ as the separator character.</td>
</tr>
<tr>
<td>'f'</td>
<td>Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.</td>
</tr>
<tr>
<td>'F'</td>
<td>Fixed-point notation. Same as 'f', but converts nan to NAN and inf to INF.</td>
</tr>
<tr>
<td></td>
<td>General format. For a given precision p &gt;= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then if -4 &lt;= exp &lt;= -1, use the exponent notation format and print the number in scientific notation using the letter ‘e’ to indicate the exponent with precision p. If exp &gt; -4, use the floating-point format with precision p. If exp &lt; -4, round the number to p significant digits and then format the result in scientific notation with an exponent that is the closest integer to exp.</td>
</tr>
</tbody>
</table>

The same as 'd'.
'g' <= exp < p, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it.

Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision.

A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'G'</td>
<td>General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.</td>
</tr>
<tr>
<td>'n'</td>
<td>Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td>'%'</td>
<td>Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.</td>
</tr>
<tr>
<td>None</td>
<td>Similar to 'g', except that fixed-point notation, when used, has at least one digit past the decimal point. The default precision is as high as needed to represent the particular value. The overall effect is to match the output of str() as altered by the other format modifiers.</td>
</tr>
</tbody>
</table>

Format examples
This section contains examples of the `str.format()` syntax and comparison with the old `%`-formatting.

In most of the cases the syntax is similar to the old `%`-formatting, with the addition of the `{}` and with : used instead of %. For example, `'%03.2f'` can be translated to `'{:03.2f}'`.

The new format syntax also supports new and different options, shown in the following examples.

Accessing arguments by position:

```python
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
a, b, c
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
a, b, c
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
c, b, a
>>> '{2}, {1}, {0}'.format(*'abc')  # unpacking
c, b, a
>>> '{0}{1}{0}'.format('abra', 'cad')  # arguments'
abracadabra
```

Accessing arguments by name:

```python
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude=37.24, longitude=-115.81)
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' attributes:

```python
>>> c = 3-5j
```
The complex number \{0\} is formed from the real part and the imaginary part \{0.imag\}. The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __str__(self):
        return 'Point({self.x}, {self.y})'.format(c)

str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```python
coord = (3, 5)
'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Replacing %s and %r:

```python
"repr() shows quotes: {!r}; str() doesn't: {!s}"
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```python
'{{:<30}}'.format('left aligned')
'left aligned'
'{{:>30}}'.format('right aligned')
'right aligned'
'{{:^30}}'.format('centered')
'centered'
'{{:*^30}}'.format('centered')  # use '*' as a fill character
'************centered************'
```
Replacing %+f, %-f, and %f and specifying a sign:

```python
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it all
'+3.140000; -3.140000'
>>> '{:f}; {:f}'.format(3.14, -3.14)  # show a space
'3.140000; -3.140000'
>>> '{:-f}; {-f}'.format(3.14, -3.14)  # show only
'3.140000; -3.140000'
```

Replacing %x and %o and converting the value to different bases:

```python
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}"
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}"
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

Using the comma as a thousands separator:

```python
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```python
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Using type-specific formatting:

```python
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
```
Nesting arguments and more complex examples:

```python
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'

>>> for align, text in zip('<>^', ['left', 'center', 'right'])
...     '{0:{fill}{align}16}'.format(text, fill=align)
... 'left<<<<<<<<<<<<'  
'^^^^^center^^^^^'
'>>>>>>>>>>>right'

>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521

>>> width = 5
>>> for num in range(5, 12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base))
...         print()
...
   5   5   5   101
   6   6   6   110
   7   7   7   111
   8   8   10  1000
   9   9   11  1001
  10  A   12  1010
  11  B   13  1011
```
Template strings

Template strings provide simpler string substitutions as described in PEP 292. A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the flufl.i18n package.

Template strings support $-based substitutions, using the following rules:

- ```$$``` is an escape; it is replaced with a single $.
- ```$identifier``` names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" is restricted to any case-insensitive ASCII alphanumeric string (including underscores) that starts with an underscore or ASCII letter. The first non-identifier character after the $ character terminates this placeholder specification.
- ```${identifier}``` is equivalent to ```$identifier```.
  It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "${noun}ification".

Any other appearance of $ in the string will result in a ValueError being raised.

The string module provides a Template class that implements these rules. The methods of Template are:

```class string.Template(template)```

The constructor takes a single argument which is the template string.
**substitute**(*mapping, **kwds*)

Performs the template substitution, returning a new string. *mapping* is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both *mapping* and *kwds* are given and there are duplicates, the placeholders from *kwds* take precedence.

**safe_substitute**(*mapping, **kwds*)

Like *substitute()*, except that if placeholders are missing from *mapping* and *kwds*, instead of raising a *KeyError* exception, the original placeholder will appear in the resulting string intact. Also, unlike with *substitute()* , any other appearances of the $ will simply return $ instead of raising *ValueError*.

While other exceptions may still occur, this method is called “safe” because it always tries to return a usable string instead of raising an exception. In another sense, *safe_substitute()* may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

**Template** instances also provide one public data attribute:

**template**

This is the object passed to the constructor’s *template* argument. In general, you shouldn’t change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```python
>>> from string import Template
```
Advanced usage: you can derive subclasses of Template to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- **delimiter** – This is the literal string describing a placeholder introducing delimiter. The default value is $. Note that this should not be a regular expression, as the implementation will call `re.escape()` on this string as needed. Note further that you cannot change the delimiter after class creation (i.e. a different delimiter must be set in the subclass's class namespace).

- **idpattern** – This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression `(?a:[_a-z][_a-z0-9]*)`. If this is given and `braceidpattern` is None this pattern will also apply to braced placeholders.
Note: Since default flags is `re.IGNORECASE`, pattern `[a-z]` can match with some non-ASCII characters. That's why we use the local a flag here.

*Changed in version 3.7:* `braceidpattern` can be used to define separate patterns used inside and outside the braces.

- `braceidpattern` – This is like `idpattern` but describes the pattern for braced placeholders. Defaults to `None` which means to fall back to `idpattern` (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

*New in version 3.7.*

- `flags` – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom `idpatterns` must follow conventions for verbose regular expressions.

*New in version 3.2.*

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute `pattern`. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- `escaped` – This group matches the escape sequence, e.g. `$$`, in the default pattern.
- `named` – This group matches the unbraced placeholder name; it
should not include the delimiter in capturing group.

- **braced** – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- **invalid** – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.
Helper functions

```python
string.capwords(s, sep=None)
```

Split the argument into words using `str.split()`, capitalize each word using `str.capitalize()`, and join the capitalized words using `str.join()`. If the optional second argument `sep` is absent or `None`, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise `sep` is used to split and join the words.
This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings (`str`) as well as 8-bit strings (`bytes`). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a byte pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `\\\` as the pattern string, because the regular expression must be `\`, and each backslash must be expressed as `\` inside a regular Python string literal.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `\` and `n`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python...
code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on compiled regular expressions. The functions are shortcuts that don’t require you to compile a regex object first, but miss some fine-tuning parameters.

**See also:** The third-party `regex` module, which has an API compatible with the standard library `re` module, but offers additional functionality and a more thorough Unicode support.
Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if $A$ and $B$ are both regular expressions, then $AB$ is also a regular expression. In general, if a string $p$ matches $A$ and another string $q$ matches $B$, the string $pq$ will match AB. This holds unless $A$ or $B$ contain low precedence operations; boundary conditions between $A$ and $B$; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [Frie09], or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'. (In the rest of this section, we'll write RE's in this special style, usually without quotes, and strings to be matched 'in single quotes'.)

Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the
regular expressions around them are interpreted.

Repetition qualifiers (*, +, ?, {m,n}, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a\{6\})* matches any multiple of six 'a' characters.

The special characters are:

.  
(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

^  
(Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

$  
Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both ‘foo’ and ‘foobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in 'foo1\nfoo2\n' matches ‘foo2’ normally, but ‘foo1’ in MULTILINE mode; searching for a single $ in 'foo\n' will find two (empty) matches: one just before the newline, and one at the end of the string.

*  
Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.
 Causes the resulting RE to match 1 or more repetitions of the preceding RE. \texttt{ab+} will match ‘a’ followed by any non-zero number of ‘b’\textquotesingle s; it will not match just ‘a’.

 Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. \texttt{ab?} will match either ‘a’ or ‘ab’.

 The ‘*’\textquotesingle, ‘+’\textquotesingle, and ‘?’\textquotesingle qualifiers are all \textit{greedy}; they match as much text as possible. Sometimes this behaviour isn\textapos;t desired; if the RE \texttt{<.*>} is matched against \texttt{<a> b <c>}, it will match the entire string, and not just \texttt{<a>}. Adding \texttt{?} after the qualifier makes it perform the match in \textit{non-greedy} or \textit{minimal} fashion; as \textit{few} characters as possible will be matched. Using the RE \texttt{<.*?>} will match only \texttt{<a>\textquotesingle}.

 Specifies that exactly \textit{m} copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, \texttt{a\{6\}} will match exactly six ‘a’ characters, but not five.

 Causes the resulting RE to match from \textit{m} to \textit{n} repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, \texttt{a\{3,5\}} will match from 3 to 5 ‘a’ characters. Omitting \textit{m} specifies a lower bound of zero, and omitting \textit{n} specifies an infinite upper bound. As an example, \texttt{a\{4,\}b} will match ‘aaaab’ or a thousand ‘a’ characters followed by a ‘b’, but not ‘aaab’. The comma may not be omitted or the modifier would be confused with the previously described form.

 \texttt{\{m, n\}?}
Causes the resulting RE to match from \(m\) to \(n\) repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', \(a\{3,5\}\) will match 5 'a' characters, while \(a\{3,5\}?\) will only match 3 characters.

Either escapes special characters (permitting you to match characters like '\\*', '\\?', and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

[]

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. \([amk]\) will match 'a', 'm', or 'k'.
- Ranges of characters can be indicated by giving two characters and separating them by a ' - ', for example \([a-z]\) will match any lowercase ASCII letter, \([0-5][0-9]\) will match all the two-digits numbers from 00 to 59, and \([0-9A-Fa-f]\) will match any hexadecimal digit. If - is escaped (e.g. \([a-\z]\)) or if it's placed as the first or last character (e.g. \([-a]\) or \([a-]\)), it will match a literal ' - '.
- Special characters lose their special meaning inside sets. For
example, `[(\+\*)]` will match any of the literal characters `'(', 
`'+'`, `'*'`, or `')'`.

- Character classes such as `\w` or `\S` (defined below) are also accepted inside a set, although the characters they match depends on whether `ASCII` or `LOCALE` mode is in force.

- Characters that are not within a range can be matched by *complementing* the set. If the first character of the set is `'^'`, all the characters that are *not* in the set will be matched. For example, `[^5]` will match any character except `'5'`, and `[^^]` will match any character except `'^'`. `^` has no special meaning if it’s not the first character in the set.

- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[(\)[\]{}]` and `[](){}[]` will both match a parenthesis.

- Support of nested sets and set operations as in Unicode Technical Standard #18 might be added in the future. This would change the syntax, so to facilitate this change a `FutureWarning` will be raised in ambiguous cases for the time being. That includes sets starting with a literal `'[` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning escape them with a backslash.

*Changed in version 3.7:* `FutureWarning` is raised if a character set contains constructs that will change semantically in the future.

\|\A\B, where \A and \B can be arbitrary REs, creates a regular expression that will match either \A or \B. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern
completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '||' operator is never greedy. To match a literal '||', use \\|, or enclose it inside a character class, as in [ || ].

(...)
Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the \number special sequence, described below. To match the literals '(' or ')', use \( or \), or enclose them inside a character class: [([], [])].

(?...) This is an extension notation (a '?' following a '()' is not meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.

(?ailmsux)
(One or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: re.A (ASCII-only matching), re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode matching), and re.X (verbose), for the entire regular expression. (The flags are described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function. Flags should be used first in the expression string.
A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.

(Zero or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x', optionally followed by '-' followed by one or more letters from the 'i', 'm', 's', 'x'.) The letters set or remove the corresponding flags: re.A (ASCII-only matching), re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode matching), and re.X (verbose), for the part of the expression. (The flags are described in Module Contents.)

The letters 'a', 'L' and 'u' are mutually exclusive when used as inline flags, so they can't be combined or follow '-'. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns (?a:...) switches to ASCII-only matching, and (?u:...) switches to Unicode matching (default). In byte pattern (?L:...) switches to locale depending matching, and (?a:...) switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

New in version 3.6.

Changed in version 3.7: The letters 'a', 'L' and 'u' also can be used in a group.
Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name `name`. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is `(\?P<quote>["'\]).*?(\?P=quote)` (i.e. matching a string quoted with either single or double quotes):

<table>
<thead>
<tr>
<th>Context of reference to group “quote”</th>
<th>Ways to reference it</th>
</tr>
</thead>
<tbody>
<tr>
<td>in the same pattern itself</td>
<td>• <code>(\?P=quote)</code></td>
</tr>
<tr>
<td></td>
<td>• \1</td>
</tr>
<tr>
<td>when processing match object <code>m</code></td>
<td>• <code>m.group('quote')</code></td>
</tr>
<tr>
<td></td>
<td>• <code>m.end('quote')</code></td>
</tr>
<tr>
<td>in a string passed to the <code>repl</code> argument of <code>re.sub()</code></td>
<td>• \g&lt;quote&gt;</td>
</tr>
<tr>
<td></td>
<td>• \g&lt;1&gt;</td>
</tr>
<tr>
<td></td>
<td>• \1</td>
</tr>
</tbody>
</table>

`(\?P=name)`
A backreference to a named group; it matches whatever text was matched by the earlier group named `name`.

`(\?#...)`
A comment; the contents of the parentheses are simply ignored.

`(\?=...)`
Matches if ... matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match 'Isaac' only if it's followed by 'Asimov'.

(?!...) Matches if ... doesn’t match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match 'Isaac' only if it's not followed by 'Asimov'.

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. (?<=abc)def will find a match in 'abcdef', since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```python
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```python
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```
Changed in version 3.5: Added support for group references of fixed length.

(?<!...)
Matches if the current position in the string is not preceded by a match for .... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?id/name)yes-pattern|no-pattern
Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn't. no-pattern is optional and can be omitted. For example, (\<)?(\w+@\w+(?:\.:\w+)+)(?(1)>|$) is a poor email matching pattern, which will match with `<user@host.com>' as well as 'user@host.com', but not with '<user@host.com' nor 'user@host.com>.'

The special sequences consist of '\' and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, \$ matches the character '$'.

\number
Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, (.+) \1 matches 'the the' or '55 55', but not 'thethe' (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value number. Inside the '[' and ']' of a character class, all numeric escapes are treated as characters.
\A
Matches only at the start of the string.

\b
Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, \b is defined as the boundary between a \w and a \W character (or vice versa), or between \w and the beginning/end of the string. This means that r'\bfoo\b' matches 'foo', 'foo.', '(foo)', 'bar foo baz' but not 'foobar' or 'foo3'.

By default Unicode alphanumerics are the ones used in Unicode patterns, but this can be changed by using the ASCII flag. Word boundaries are determined by the current locale if the LOCALE flag is used. Inside a character range, \b represents the backspace character, for compatibility with Python’s string literals.

\B
Matches the empty string, but only when it is not at the beginning or end of a word. This means that r'py\B' matches 'python', 'py3', 'py2', but not 'py', 'py.', or 'py!'. \B is just the opposite of \b, so word characters in Unicode patterns are Unicode alphanumerics or the underscore, although this can be changed by using the ASCII flag. Word boundaries are determined by the current locale if the LOCALE flag is used.

\d
For Unicode (str) patterns:
Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9], and also many other digit characters. If the ASCII flag is used only [0-9] is matched.
For 8-bit (bytes) patterns:

Matches any decimal digit; this is equivalent to \[0-9\].

\D

Matches any character which is not a decimal digit. This is the opposite of \d. If the ASCII flag is used this becomes the equivalent of [^0-9].

\s

For Unicode (str) patterns:

Matches Unicode whitespace characters (which includes [\t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the ASCII flag is used, only [\t\n\r\f\v] is matched.

For 8-bit (bytes) patterns:

Matches characters considered whitespace in the ASCII character set; this is equivalent to [\t\n\r\f\v].

\S

Matches any character which is not a whitespace character. This is the opposite of \s. If the ASCII flag is used this becomes the equivalent of [^\t\n\r\f\v].

\w

For Unicode (str) patterns:

Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the ASCII flag is used, only [a-zA-Z0-9_] is matched.

For 8-bit (bytes) patterns:

Matches characters considered alphanumeric in the ASCII character set; this is equivalent to [a-zA-Z0-9_]. If the
The `locale` flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W
Matches any character which is not a word character. This is the opposite of \w. If the `ASCII` flag is used this becomes the equivalent of \^[a-zA-Z0-9_]\. If the `locale` flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\Z
Matches only at the end of the string.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

<table>
<thead>
<tr>
<th>\a</th>
<th>\b</th>
<th>\f</th>
<th>\n</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r</td>
<td>\t</td>
<td>\u</td>
<td>\U</td>
</tr>
<tr>
<td>\v</td>
<td>\x</td>
<td>\</td>
<td></td>
</tr>
</tbody>
</table>

(Note that \b is used to represent word boundaries, and means “backspace” only inside character classes.)

`\u` and `\U` escape sequences are only recognized in Unicode patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

*Changed in version 3.3:* The `\u` and `\U` escape sequences have been added.
*Changed in version 3.6:* Unknown escapes consisting of `'\'` and an ASCII letter now are errors.
Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

*Changed in version 3.6:* Flag constants are now instances of RegexFlag, which is a subclass of `enum.IntFlag`.

**re.compile(pattern, flags=0)**

Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods, described below.

The expression's behaviour can be modified by specifying a `flags` value. Values can be any of the following variables, combined using bitwise OR (the `|` operator).

The sequence

```python
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```python
result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.
Note: The compiled versions of the most recent patterns passed to \texttt{re.compile()} and the module-level matching functions are cached, so programs that use only a few regular expressions at a time needn’t worry about compiling regular expressions.

\texttt{re.A}

\texttt{re.ASCII}

Make \texttt{\w, \W, \b, \B, \d, \D, \s} and \texttt{\S} perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns. Corresponds to the inline flag (?a).

Note that for backward compatibility, the \texttt{re.U} flag still exists (as well as its synonym \texttt{re.UNICODE} and its embedded counterpart (?u)), but these are redundant in Python 3 since matches are Unicode by default for strings (and Unicode matching isn’t allowed for bytes).

\texttt{re.DEBUG}

Display debug information about compiled expression. No corresponding inline flag.

\texttt{re.I}

\texttt{re.IGNORECASE}

Perform case-insensitive matching; expressions like \texttt{[A-Z]} will also match lowercase letters. Full Unicode matching (such as Ü matching ü) also works unless the \texttt{re.ASCII} flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the \texttt{re.LOCALE} flag is also used. Corresponds to the inline flag (?i).

Note that when the Unicode patterns \texttt{[a-z]} or \texttt{[A-Z]} are used in
combination with the **IGNORECASE** flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: ‘ı’ (U+0130, Latin capital letter I with dot above), ‘ı’ (U+0131, Latin small letter dotless i), ‘ſ’ (U+017F, Latin small letter long s) and ‘K’ (U+212A, Kelvin sign). If the **ASCII** flag is used, only letters ‘a’ to ‘z’ and ‘A’ to ‘Z’ are matched.

**re.L**

**re.LOCALE**

Make `\w, \W, \b, \B` and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns. The use of this flag is discouraged as the locale mechanism is very unreliable, it only handles one “culture” at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages. Corresponds to the inline flag `(?L)`.

*Changed in version 3.6:* **re.LOCALE** can be used only with bytes patterns and is not compatible with **re.ASCII**.

*Changed in version 3.7:* Compiled regular expression objects with the **re.LOCALE** flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

**re.M**

**re.MULTILINE**

When specified, the pattern character `\^` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `\$` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `\^` matches
only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string. Corresponds to the inline flag (\(?m\)).

re.S
re.DOTALL
Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything except a newline. Corresponds to the inline flag (\(?s\)).

re.X
re.VERBOSE
This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like *?, (?: or (?P<...>). When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```python
a = re.compile(r"\d+ # the integral part
  \. # the decimal point
  \d* # some fractional digits"
)
b = re.compile(r"\d+\.\d*"
)
```

Corresponds to the inline flag (\(?x\)).
**re.search**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding *match object*. Return **None** if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

**re.match**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding *match object*. Return **None** if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in **MULTILINE** mode, **re.match()** will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in *string*, use **search()** instead (see also **search() vs. match()**).

**re.fullmatch**(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding *match object*. Return **None** if the string does not match the pattern; note that this is different from a zero-length match.

*New in version 3.4.*

**re.split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.
If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```python
>>> re.split(r'\W+', 'Words, words, words."
['Words', 'words', 'words', '"
```
re. **findall**(pattern, string, flags=0)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

*Changed in version 3.7:* Non-empty matches can now start just after a previous empty match.

re. **finditer**(pattern, string, flags=0)

Return an iterator yielding *match objects* over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

*Changed in version 3.7:* Non-empty matches can now start just after a previous empty match.

re. **sub**(pattern, repl, string, count=0, flags=0)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn’t found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:
If `repl` is a function, it is called for every non-overlapping occurrence of `pattern`. The function takes a single `match object` argument, and returns the replacement string. For example:

```python
>>> def dashrepl(matchobj):
...     if matchobj.group(1) == '-':
...         return ''
...     else:
...         return '-'

>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'

>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam')
'Baked Beans & Spam'
```

The pattern may be a string or a `pattern object`.

The optional argument `count` is the maximum number of pattern occurrences to be replaced; `count` must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous empty match, so `sub('x*', '-', 'abxd')` returns `-a-b--d-`.

In string-type `repl` arguments, in addition to the character escapes and backreferences described above, \g<name> will use the substring matched by the group named name, as defined by the (? P<name>...) syntax. \g<number> uses the corresponding group number; \g<2> is therefore equivalent to \2, but isn't ambiguous in a replacement such as \g<2>0. \20 would be
interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'. The backreference \g<0> substitutes in the entire substring matched by the RE.

*Changed in version 3.1:* Added the optional flags argument.

*Changed in version 3.5:* Unmatched groups are replaced with an empty string.

*Changed in version 3.6:* Unknown escapes in *pattern* consisting of '\' and an ASCII letter now are errors.

*Changed in version 3.7:* Unknown escapes in *repl* consisting of '\' and an ASCII letter now are errors.

Empty matches for the pattern are replaced when adjacent to a previous non-empty match.

**re.subn**(pattern, repl, string, count=0, flags=0)

Perform the same operation as *sub()*, but return a tuple (new_string, number_of_subs_made).

*Changed in version 3.1:* Added the optional flags argument.

*Changed in version 3.5:* Unmatched groups are replaced with an empty string.

**re.escape**(pattern)

Escape special characters in *pattern*. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```python
>>> print(re.escape('python.exe'))
python\.exe
```
legal_chars = string.ascii_lowercase + string.ascii_lowercase
print('[%s]+' % re.escape(legal_chars))

operators = ['+', '-', '*', '/', '***']
print('|'.join(map(re.escape, sorted(operators))))

This functions must not be used for the replacement string in sub() and subn(), only backslashes should be escaped. For example:

digits_re = r'\d+'
sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
print(re.sub(digits_re, digits_re.replace('\\', '/usr/sbin/sendmail - \d+ errors, \d+ warnings

Changed in version 3.3: The '_' character is no longer escaped.

Changed in version 3.7: Only characters that can have special meaning in a regular expression are escaped.

re.purge()
Clear the regular expression cache.

exception re.error(msg, pattern=None, pos=None)
Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern. The error instance has the following additional attributes:
msg
   The unformatted error message.

pattern
   The regular expression pattern.

pos
   The index in pattern where compilation failed (may be None).

lineno
   The line corresponding to pos (may be None).

colno
   The column corresponding to pos (may be None).

Changed in version 3.5: Added additional attributes.
Regular Expression Objects

Compiled regular expression objects support the following methods and attributes:

```python
Pattern.search(string[, pos[, endpos]])
```

Scan through `string` looking for the first location where this regular expression produces a match, and return a corresponding match object. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter `pos` gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter `endpos` limits how far the string will be searched; it will be as if the string is `endpos` characters long, so only the characters from `pos` to `endpos - 1` will be searched for a match. If `endpos` is less than `pos`, no match will be found; otherwise, if `rx` is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```python
>>> pattern = re.compile("d")
>>> pattern.search("dog")  # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search
```
Pattern. **match**(string[, pos[, endpos]])

If zero or more characters at the *beginning of string* match this regular expression, return a corresponding *match object*. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the **search()** method.

```python
>>> pattern = re.compile("o")
>>> pattern.match("dog")    # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)  # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use **search()** instead (see also **search() vs. match()**).

Pattern. **fullmatch**(string[, pos[, endpos]])

If the whole *string* matches this regular expression, return a corresponding *match object*. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the **search()** method.

```python
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")    # No match as "og" is not at the start of "dog".
>>> pattern.fullmatch("ogre")   # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```
New in version 3.4.

Pattern. **split**(string, maxsplit=0)
Identical to the `split()` function, using the compiled pattern.

Pattern. **findall**(string[, pos[, endpos]])
Similar to the `findall()` function, using the compiled pattern, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.

Pattern. **finditer**(string[, pos[, endpos]])
Similar to the `finditer()` function, using the compiled pattern, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.

Pattern. **sub**(repl, string, count=0)
Identical to the `sub()` function, using the compiled pattern.

Pattern. **subn**(repl, string, count=0)
Identical to the `subn()` function, using the compiled pattern.

Pattern. **flags**
The regex matching flags. This is a combination of the flags given to `compile()`, any `(?...)` inline flags in the pattern, and implicit flags such as UNICODE if the pattern is a Unicode string.

Pattern. **groups**
The number of capturing groups in the pattern.

Pattern. **groupindex**
A dictionary mapping any symbolic group names defined by `(?P<id>)` to group numbers. The dictionary is empty if no symbolic
groups were used in the pattern.

**Pattern**. *pattern*
The pattern string from which the pattern object was compiled.

*Changed in version 3.7*: Added support of `copy.copy()` and `copy.deepcopy()`. Compiled regular expression objects are considered atomic.
Match Objects

Match objects always have a boolean value of True. Since `match()` and `search()` return None when there is no match, you can test whether there was a match with a simple `if` statement:

```python
match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support the following methods and attributes:

**Match. `expand(template)`**

Return the string obtained by doing backslash substitution on the template string `template`, as done by the `sub()` method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

*Changed in version 3.5:* Unmatched groups are replaced with an empty string.

**Match. `group([group1,...])`**

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, `group1` defaults to zero (the whole match is returned). If a `groupN` argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized
group. If a group number is negative or larger than the number of groups defined in the pattern, an *IndexError* exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```python
group.

>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)  # The entire match
'Isaac Newton'
>>> m.group(1)  # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)  # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)  # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an *IndexError* exception is raised.

A moderately complicated example:

```python
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```python
>>> m.group(1)
```

'Malcolm'

>>> m.group(2)
'Reynolds'

If a group matches multiple times, only the last match is accessible:

```python
>>> m = re.match(r"(\..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)  # Returns only the last match.
'c3'
```

**Match.\_getitem\_** (g)

This is identical to `m.group(g)`. This allows easier access to an individual group from a match:

```python
>>> m = re.match(r"(\w+) \(\w+\)", "Isaac Newton, physicist")
>>> m[0]  # The entire match
'Isaac Newton'
>>> m[1]  # The first parenthesized subgroup
'Isaac'
>>> m[2]  # The second parenthesized subgroup
'Newton'
```

*New in version 3.6.*

**Match.\_groups\_** (default=None)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The `default` argument is used for groups that did not participate in the match; it defaults to `None`.

For example:

```python
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
```
If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

```python
>>> m = re.match(r"\d+.?\d+", "24")
>>> m.groups()  # Second group defaults to None
('24', None)
>>> m.groups('0')  # Now, the second group defaults to '0'
('24', '0')
```

**Match.groupdict**(default=None)

Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None. For example:

```python
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

**Match.start**([group])

**Match.end**([group])

Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring). Return -1 if group exists but did not contribute to the match. For a match object m, and a group g that did contribute to the match, the substring matched by group g (equivalent to m.group(g)) is
Note that `m.start(group)` will equal `m.end(group)` if `group` matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an `IndexError` exception.

An example that will remove `remove_this` from email addresses:

```python
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

**Match. `span`**

For a match `m`, return the 2-tuple `(m.start(group), m.end(group))`. Note that if `group` did not contribute to the match, this is `(-1, -1)`. `group` defaults to zero, the entire match.

**Match. `pos`**

The value of `pos` which was passed to the `search()` or `match()` method of a `regex object`. This is the index into the string at which the RE engine started looking for a match.

**Match. `endpos`**

The value of `endpos` which was passed to the `search()` or `match()` method of a `regex object`. This is the index into the string beyond which the RE engine will not go.

**Match. `lastindex`**

The integer index of the last matched capturing group, or `None` if
no group was matched at all. For example, the expressions \((a)b,(a)(b))\, and \(((ab))\) will have `lastindex == 1` if applied to the string `"ab"`, while the expression \((a)(b)\) will have `lastindex == 2`, if applied to the same string.

**Match. lastgroup**

The name of the last matched capturing group, or `None` if the group didn’t have a name, or if no group was matched at all.

**Match. re**

The regular expression object whose `match()` or `search()` method produced this match instance.

**Match. string**

The string passed to `match()` or `search()`.

*Changed in version 3.7:* Added support of `copy.copy()` and `copy.deepcopy()`. Match objects are considered atomic.*
Regular Expression Examples

Checking for a Pair

In this example, we'll use the following helper function to display match objects a little more gracefully:

```python
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(),)
```

Suppose you are writing a poker program where a player's hand is represented as a 5-character string with each character representing a card, “a” for ace, “k” for king, “q” for queen, “j” for jack, “t” for 10, and “2” through “9” representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```python
>>> valid = re.compile(r'^[a2-9tjqq]{5}$')
>>> displaymatch(valid.match("akt5q"))  # Valid.
'<Match: 'akt5q', groups=()>'
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))  # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
'<Match: '727ak', groups=()>'
```

That last hand, "727ak", contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```python
>>> pair = re.compile(r".\(.\).*\1")
>>> displaymatch(pair.match("717ak"))  # Pair of
To find out what card the pair consists of, one could use the `group()` method of the match object in the following manner:

```python
>>> pair.match("717ak").group(1)
'7'
```

# Error because re.match() returns None, which doesn't have a group() method:

```python
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(\.)\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

```python
>>> pair.match("354aa").group(1)
'a'
```

## Simulating scanf()

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

<table>
<thead>
<tr>
<th><code>scanf()</code> Token</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c</code></td>
<td>.</td>
</tr>
<tr>
<td><code>%5c</code></td>
<td>.{5}</td>
</tr>
</tbody>
</table>
To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

**search() vs. match()**

Python offers two different primitive operations based on regular expressions: `re.match()` checks for a match only at the beginning of the string, while `re.search()` checks for a match anywhere in the string (this is what Perl does by default).

For example:
Regular expressions beginning with `^` can be used with `search()` to restrict the match at the beginning of the string:

```python
>>> re.match("c", "abcdef")  # No match
>>> re.search("^c", "abcdef")  # Match
<re.Match object; span=(2, 3), match='c'>
```

Note however that in `MULTILINE` mode `match()` only matches at the beginning of the string, whereas using `search()` with a regular expression beginning with `^` will match at the beginning of each line.

```python
>>> re.match("c", "abcdef")  # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Making a Phonebook

`split()` splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```python
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street""
```
The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```python
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street', 'Ronald Heathmore: 892.345.3428 436 Finley Avenue', 'Frank Burger: 925.541.7625 662 South Dogwood Way', 'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the maxsplit parameter of `split()` because the address has spaces, our splitting pattern, in it:

```python
>>> [re.split(":?:", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'], ['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'], ['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'], ['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The `:?:` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of 4, we could separate the house number from the street name:

```python
>>> [re.split(":?:", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'], ['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue']]
```
Text Munging

`sub()` replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using `sub()` with a function to “munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```python
>>> def repl(m):
...    inner_word = list(m.group(2))
...    random.shuffle(inner_word)
...    return m.group(1) + ''.join(inner_word) + m

>>> text = "Professor Abdolmalek, please report your absences promptly."

>>> re.sub(r"(\w)(\w+)(\w)\w+ly", repl, text)
'Poefsrosr Aealmlobdk, pslaee reoerpt your abnseces p

>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces p
```

Finding all Adverbs

`findall()` matches all occurrences of a pattern, not just the first one as `search()` does. For example, if a writer wanted to find all of the adverbs in some text, they might use `findall()` in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."

>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```
Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, `finditer()` is useful as it provides match objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs and their positions in some text, they would use `finditer()` in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), 07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (r"text") keeps regular expressions sane. Without it, every backslash ('\') in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```python
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means r"\". Without raw string notation, one must use "\\", making the following lines of code functionally identical:
Writing a Tokenizer

A tokenizer or scanner analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```python
import collections
import re

Token = collections.namedtuple('Token', ['type', 'value'])

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT'}
    token_specification = [
        ('NUMBER', r'\d+(\.\d*)?'),                         # Integer or decimal number
        ('ASSIGN', r'\='),                                  # Assignment operator
        ('END', r';'),                                      # Statement terminator
        ('ID', r'[A-Za-z]+'),                               # Identifiers
        ('OP', r'[\+\-*/]'),                               # Arithmetic operators
        ('NEWLINE', r'\n'),                                # Line endings
        ('SKIP', r'[ \t]+'),                                # Skip over spaces and tabs
        ('MISMATCH', r'.'),                                # Any other character
    ]
    tok_regex = '|'.join('\b{0}.\b'.format(s) for s, t in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        if mo.lastgroup == 'MISMATCH':
            return
        if mo.lastgroup in keywords:
            if mo.lastgroup == 'END':
                line_num += 1
                line_start = 0
            line_start = 0
```
kind = mo.lastgroup
value = mo.group()
column = mo.start() - line_start
if kind == 'NUMBER':
    value = float(value) if '.' in value else
elif kind == 'ID' and value in keywords:
    kind = value
elif kind == 'NEWLINE':
    line_start = mo.end()
    line_num += 1
    continue
elif kind == 'SKIP':
    continue
elif kind == 'MISMATCH':
    raise RuntimeError(f'{value!r} unexpected')
yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
...'''

for token in tokenize(statements):
    print(token)

The tokenizer produces the following output:

Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
difflib — Helpers for computing deltas

Source code: Lib/difflib.py

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the filecmp module.

class difflib.SequenceMatcher

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershlp under the hyperbolic name “gestalt pattern matching.” The idea is to find the longest contiguous matching subsequence that contains no “junk” elements; these “junk” elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershlp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.

Timing: The basic Ratcliff-Obershlp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has
expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

**Automatic junk heuristic:** `SequenceMatcher` supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item’s duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as “popular” and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the `autojunk` argument to `False` when creating the `SequenceMatcher`.

*New in version 3.2:* The `autojunk` parameter.

**class** `difflib.Differ`

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses `SequenceMatcher` both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a `Differ` delta begins with a two-letter code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>line unique to sequence 1</td>
</tr>
<tr>
<td>'+'</td>
<td>line unique to sequence 2</td>
</tr>
<tr>
<td>' '</td>
<td>line common to both sequences</td>
</tr>
<tr>
<td>'?'</td>
<td>line not present in either input sequence</td>
</tr>
</tbody>
</table>
Lines beginning with ‘?’ attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

```python
class difflib.HtmlDiff
```

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

```python
__init__(tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK)
```

Initializes instance of `HtmlDiff`.

- `tabsize` is an optional keyword argument to specify tab stop spacing and defaults to 8.
- `wrapcolumn` is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.
- `linejunk` and `charjunk` are optional keyword arguments passed into `ndiff()` (used by `HtmlDiff` to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

```python
make_file(fromlines, tolines, fromdesc='', todesc='', context=False, numlines=5, *, charset='utf-8')
```
Compares `fromlines` and `tolines` (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

`fromdesc` and `todesc` are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

`context` and `numlines` are both optional keyword arguments. Set `context` to `True` when contextual differences are to be shown, else the default is `False` to show the full files. `numlines` defaults to `5`. When `context` is `True` `numlines` controls the number of context lines which surround the difference highlights. When `context` is `False` `numlines` controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

*Changed in version 3.5:* `charset` keyword-only argument was added. The default charset of HTML document changed from `'ISO-8859-1'` to `'utf-8'`.

### make_table

```python
make_table(fromlines, tolines, fromdesc='', todesc='', context=False, numlines=5)
```

Compares `fromlines` and `tolines` (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the `make_file()` method.
Tools/scripts/diff.py is a command-line front-end to this class and contains a good example of its use.

difflib.

**context_diff**(a, b, fromfile='', tofile='', fromfiledate='', tofiledate='', n=3, lineterm='\n')

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with *** or ---) are created with a trailing newline. This is helpful so that inputs created from io.IOBase.readlines() result in diffs that are suitable for use with io.IOBase.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n'
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido

>>> sys.stdout.writelines(context_diff(s1, s2, fr
*** before.py
--- after.py
***************
```
See A command-line interface to difflib for a more detailed example.

difflib.get_close_matches(word, possibilities, n=3, cutoff=0.6)

Return a list of the best “good enough” matches. word is a sequence for which close matches are desired (typically a string), and possibilities is a list of sequences against which to match word (typically a list of strings).

Optional argument n (default 3) is the maximum number of close matches to return; n must be greater than 0.

Optional argument cutoff (default 0.6) is a float in the range [0, 1]. Possibilities that don’t score at least that similar to word are ignored.

The best (no more than n) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```python
>>> get_close_matches('appel', ['ape', 'apple', 'peach'], ['apple', 'ape'])
>>> import keyword
```
get_close_matches('wheel', keyword.kwlist)
['while']
get_close_matches('pineapple', keyword.kwlist)
[]
get_close_matches('accept', keyword.kwlist)
['except']

difflib.ndiff(a, b, linejunk=None, charjunk=IS_CHARACTER_JUNK)

Compare a and b (lists of strings); return a Differ-style delta (a generator generating the delta lines).

Optional keyword parameters linejunk and charjunk are filtering functions (or None):

*linejunk*: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is None. There is also a module-level function IS_LINE_JUNK(), which filters out lines without visible characters, except for at most one pound character ('#') – however the underlying SequenceMatcher class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

*charjunk*: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function IS_CHARACTER_JUNK(), which filters out whitespace characters (a blank or tab; it's a bad idea to include newline in this!).

Tools/scripts/ndiff.py is a command-line front-end to this function.
```python
diff = ndiff('one\ntwo\nthree
'.splitlines(k ...
    'ore\ntree\nemu
'.splitlines(k...
>>> print(''.join(diff), end='')
- one
  + ore
  - two
- three
  + tree
+ emu
```
difflib.unified_diff(a, b, fromfile='', tofile='', fromfiledate='', tofiledate='', n=3, lineterm='\n')

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with ---, +++, or @@) are created with a trailing newline. This is helpful so that inputs created from io.IOBase.readlines() result in diffs that are suitable for use with io.IOBase.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n'
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'gui
>>> sys.stdout.writelines(unified_diff(s1, s2, fr
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
```
See A command-line interface to difflib for a more detailed example.

```python
difflib.diff_bytes(dfunc, a, b, fromfile=b'', tofile=b'', fromfiledate=b'', tofiledate=b'', n=3, lineterm=b'\n')
```

Compare a and b (lists of bytes objects) using dfunc; yield a sequence of delta lines (also bytes) in the format returned by dfunc. `dfunc` must be a callable, typically either `unified_diff()` or `context_diff()`.

Allows you to compare data with unknown or inconsistent encoding. All inputs except n must be bytes objects, not str. Works by losslessly converting all inputs (except n) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of dfunc is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as a and b.

New in version 3.5.

```python
difflib.IS_LINE_JUNK(line)
```

Return true for ignorable lines. The line `line` is ignorable if `line` is blank or contains a single '##', otherwise it is not ignorable. Used as a default for parameter `linejunk` in `ndiff()` in older versions.

```python
difflib.IS_CHARACTER_JUNK(ch)
```

Return true for ignorable characters. The character `ch` is ignorable if `ch` is a space or tab, otherwise it is not ignorable. Used as a
default for parameter \textit{charjunk} in \texttt{ndiff()}. 

\textbf{See also:}

\textbf{Pattern Matching: The Gestalt Approach}

Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in \textit{Dr. Dobb's Journal} in July, 1988.
SequenceMatcher Objects

The `SequenceMatcher` class has this constructor:

```python
class difflib.SequenceMatcher(isjunk=None, a='', b='', autojunk=True)
```

Optional argument `isjunk` must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is “junk” and should be ignored. Passing `None` for `isjunk` is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```python
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments `a` and `b` are sequences to be compared; both default to empty strings. The elements of both sequences must be hashable.

The optional argument `autojunk` can be used to disable the automatic junk heuristic.

*New in version 3.2*: The `autojunk` parameter.

SequenceMatcher objects get three data attributes: `bjunk` is the set of elements of `b` for which `isjunk` is `True`; `bpopular` is the set of non-junk elements considered popular by the heuristic (if it is not disabled); `b2j` is a dict mapping the remaining elements of `b` to a list of positions where they occur. All three are reset whenever `b` is reset with `set_seqs()` or `set_seq2()`.
New in version 3.2: The `bjunk` and `bpopular` attributes.

**SequenceMatcher** objects have the following methods:

**set_seqs**(a, b)
- Set the two sequences to be compared.

*SequenceMatcher* computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use `set_seq2()` to set the commonly used sequence once and call `set_seq1()` repeatedly, once for each of the other sequences.

**set_seq1**(a)
- Set the first sequence to be compared. The second sequence to be compared is not changed.

**set_seq2**(b)
- Set the second sequence to be compared. The first sequence to be compared is not changed.

**find_longest_match**(alo, ahi, blo, bhi)
- Find longest matching block in a[alo:ahi] and b[blo:bhi].

If `isjunk` was omitted or `None`, `find_longest_match()` returns `(i, j, k)` such that a[i:i+k] is equal to b[j:j+k], where alo <= i <= i+k <= ahi and blo <= j <= j+k <= bhi. For all (i', j', k') meeting those conditions, the additional conditions k >= k', i <= i', and if i == i', j <= j' are also met. In other words, of all maximal matching blocks, return one that starts earliest in a,
and of all those maximal matching blocks that start earliest in $a$, return the one that starts earliest in $b$.

```python
>>> s = SequenceMatcher(None, "abcd", "abcd abc
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If `isjunk` was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here’s the same example as before, but considering blanks to be junk. That prevents 'abcd' from matching the 'abcd' at the tail end of the second sequence directly. Instead only the 'abcd' can match, and matches the leftmost 'abcd' in the second sequence:

```python
>>> s = SequenceMatcher(lambda x: x==" ", " abc
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns $(a0, b0, 0)$.

This method returns a named tuple `Match(a, b, size)`. 

**get_matching_blocks()**

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form $(i, j, n)$, and means that $a[i:i+n] == b[j:j+n]$. The triples are
monotonically increasing in $i$ and $j$.

The last triple is a dummy, and has the value $(\text{len}(a), \text{len}(b), 0)$. It is the only triple with $n == 0$. If $(i, j, n)$ and $(i', j', n')$ are adjacent triples in the list, and the second is not the last triple in the list, then $i+n < i'$ or $j+n < j'$; in other words, adjacent triples always describe non-adjacent equal blocks.

```python
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

**get_opcodes()**

Return list of 5-tuples describing how to turn $a$ into $b$. Each tuple is of the form $(\text{tag}, i_1, i_2, j_1, j_2)$. The first tuple has $i_1 == j_1 == 0$, and remaining tuples have $i_1$ equal to the $i_2$ from the preceding tuple, and, likewise, $j_1$ equal to the previous $j_2$.

The *tag* values are strings, with these meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'replace'</td>
<td>$a[i_1:i_2]$ should be replaced by $b[j_1:j_2]$.</td>
</tr>
<tr>
<td>'delete'</td>
<td>$a[i_1:i_2]$ should be deleted. Note that $j_1 == j_2$ in this case.</td>
</tr>
<tr>
<td>'insert'</td>
<td>$b[j_1:j_2]$ should be inserted at $a[i_1:i_1]$. Note that $i_1 == i_2$ in this case.</td>
</tr>
<tr>
<td>'equal'</td>
<td>$a[i_1:i_2] == b[j_1:j_2]$ (the sub-sequences are equal).</td>
</tr>
</tbody>
</table>
For example:

```python
global

>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...   print('{:7} a[{:}{:}] --> b[{:}{:}] {!r}
...     delete a[0:1] --> b[0:0]   'q' --> ''
equal a[1:3] --> b[0:2]   'ab' --> 'ab'
```

**get_grouped_opcodes**($n=3$)

Return a generator of groups with up to $n$ lines of context.

Starting with the groups returned by **get_opcodes()**, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as **get_opcodes()**.

**ratio()**

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where $T$ is the total number of elements in both sequences, and $M$ is the number of matches, this is $2.0*\frac{M}{T}$. Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.
This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn’t already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

**quick_ratio()**

Return an upper bound on `ratio()` relatively quickly.

**real_quick_ratio()**

Return an upper bound on `ratio()` very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()`:

```python
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```
SequenceMatcher Examples

This example compares two strings, considering blanks to be “junk”:

```python
>>> s = SequenceMatcher(lambda x: x == ' ',
                       "private Thread currentThread;
                       "private volatile Thread cur...
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches:

```python
>>> print(round(s.ratio(), 3))
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```python
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" %
           (block.a_start, block.b_start, len(block.matched_string)))
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```python
>>> for opcode in s.get_opcodes():
```
See also:

- The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
- Simple version control recipe for a small application built with `SequenceMatcher`.
Differ Objects

Note that Differ-generated deltas make no claim to be minimal diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The Differ class has this constructor:

```python
class difflib.Differ(linejunk=None, charjunk=None)
```

Optional keyword parameters `linejunk` and `charjunk` are for filter functions (or None):

- **`linejunk`**: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

- **`charjunk`**: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the `find_longest_match()` method's `isjunk` parameter for an explanation.

Differ objects are used (deltas generated) via a single method:

```python
compare(a, b)
```

Compare two sequences of lines, and generate the delta (a
sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.
Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```python
>>> text1 = ''' 1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
...   3. Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''
```

Next we instantiate a Differ object:

```python
>>> d = Differ()
```

Note that when instantiating a `Differ` object we may pass functions to filter out line and character “junk.” See the `Differ()` constructor for details.

Finally, we compare the two:

```python
>>> result = list(d.compare(text1, text2))
```
result is a list of strings, so let's pretty-print it:

```python
>>> from pprint import pprint
>>> pprint(result)
['  1. Beautiful is better than ugly.
', '-  2. Explicit is better than implicit.
', '-  3. Simple is better than complex.
', '+  3. Simple is better than complex.
', '?   ++
', '-  4. Complex is better than complicated.
', '?      ^
', '+  4. Complicated is better than complex.
', '+  5. Flat is better than nested.
]
```

As a single multi-line string it looks like this:

```python
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
?   ++
- 4. Complex is better than complicated.
?      ^
+ 4. Complicated is better than complex.
?      ++++
+ 5. Flat is better than nested.
```
A command-line interface to difflib

This example shows how to use difflib to create a diff-like utility. It is also contained in the Python source distribution, as Tools/scripts/diff.py.

```python
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:
* ndiff: lists every line and highlights interline changes.
* context: highlights clusters of changes in a before/after format.
* unified: highlights clusters of changes in an inline format.
* html: generates side by side comparison with change highlights.
""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime)
    return t.astimezone().isoformat()

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true',
                        help='Produce a context format (can use -c and -l in conjunction)')
    parser.add_argument('-u', action='store_true',
                        help='Produce a unified format')
    parser.add_argument('-m', action='store_true',
                        help='Produce HTML side by side comparison')
    parser.add_argument('-n', action='store_true',
                        help='Produce HTML side by side comparison')
```
help='Produce a ndiff format'
parser.add_argument('-l', '--lines', type=int, default=3, help='Set number of context lines (default 3)'
parser.add_argument('fromfile')
parser.add_argument('tofile')
options = parser.parse_args()

n = options.lines
fromfile = options.fromfile
tofile = options.tofile

fromdate = file_mtime(fromfile)
todate = file_mtime(tofile)
with open(fromfile) as ff:
    fromlines = ff.readlines()
with open(tofile) as tf:
    tolines = tf.readlines()

if options.u:
    diff = difflib.unified_diff(fromlines, tolines)
elif options.n:
    diff = difflib.ndiff(fromlines, tolines)
elif options.m:
    diff = difflib.HtmlDiff().make_file(fromlines, tolines)
else:
    diff = difflib.context_diff(fromlines, tolines)
sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
textwrap — Text wrapping and filling

Source code: Lib/textwrap.py

The textwrap module provides some convenience functions, as well as TextWrapper, the class that does all the work. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of TextWrapper for efficiency.

textwrap.wrap(text, width=70, **kwargs)

Wraps the single paragraph in text (a string) so every line is at most width characters long. Returns a list of output lines, without final newlines.

Optional keyword arguments correspond to the instance attributes of TextWrapper, documented below. width defaults to 70.

See the TextWrapper.wrap() method for additional details on how wrap() behaves.

textwrap.fill(text, width=70, **kwargs)

Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph. fill() is shorthand for

"\n".join(wrap(text, ...))

In particular, fill() accepts exactly the same keyword arguments as wrap().
textwrap. **shorten**(*text, width, **kwargs*)

Collapse and truncate the given *text* to fit in the given *width*.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the placeholder fit within *width*:

```python
>>> textwrap.shorten("Hello world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world!", width=10, placeholder='Hello...')

```

Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below. Note that the whitespace is collapsed before the text is passed to the `TextWrapper fill()` function, so changing the value of `tabsize`, `expand_tabs`, `drop_whitespace`, and `replace_whitespace` will have no effect.

New in version 3.4.

textwrap. **dedent**(*text*)

Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines "hello" and "\thello" are
considered to have no common leading whitespace.

For example:

```python
def test():
    # end first line with \ to avoid the empty line!
    s = '''
    hello
    ...'
    print(repr(s))  # prints 'hello
    print(repr(dedent(s)))  # prints 'hello
    ```
textwrap.\indent(text, prefix, predicate=None)

Add prefix to the beginning of selected lines in text.

Lines are separated by calling \text.splitlines(True).

By default, prefix is added to all lines that do not consist solely of whitespace (including any line endings).

For example:

```python
>>> s = 'hello


world'
>>> indent(s, '  ')
  hello
  hello
  world'
```

The optional predicate argument can be used to control which lines are indented. For example, it is easy to add prefix to even empty and whitespace-only lines:

```python
>>> print(indent(s, '+ ', \lambda line: True))
+ hello
+ +
```
wrap(), fill() and shorten() work by creating a TextWrapper instance and calling a single method on it. That instance is not reused, so for applications that process many text strings using wrap() and/or fill(), it may be more efficient to create your own TextWrapper object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless TextWrapper.break_long_words is set to false.

class textwrap.TextWrapper(**kwargs)

The TextWrapper constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```
wrapper = TextWrapper()
wrapper.initial_indent = "* 
```

You can re-use the same TextWrapper object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The TextWrapper instance attributes (and keyword arguments to the constructor) are as follows:

+ world
**width**
(default: 70) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than `width`, `TextWrapper` guarantees that no output line will be longer than `width` characters.

**expand_tabs**
(default: True) If true, then all tab characters in `text` will be expanded to spaces using the `expandtabs()` method of `text`.

**tabsize**
(default: 8) If `expand_tabs` is true, then all tab characters in `text` will be expanded to zero or more spaces, depending on the current column and the given tab size.

*New in version 3.3.*

**replace_whitespaces**
(default: True) If true, after tab expansion but before wrapping, the `wrap()` method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`\t\n\v\f\r`).

**Note:** If `expand_tabs` is false and `replace_whitespaces` is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

**Note:** If `replace_whitespaces` is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using `str.splitlines()` or similar) which are wrapped
drop_whitespace
(default: True) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

initial_indent
(default: '') String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent
(default: '') String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings
(default: False) If true, TextWrapper attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of '.', '!', or '?', possibly followed by one of '"' or '"', followed by a space. One problem with this is algorithm is that it is unable to detect the difference between “Dr.” in

[[... Dr. Frankenstein's monster [[...]]]
and “Spot.” in

```
[...] See Spot. See Spot run [...] 
```

`fix_sentence_endings` is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter,” and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

**break_long_words**
(default: `True`) If true, then words longer than `width` will be broken in order to ensure that no lines are longer than `width`. If it is false, long words will not be broken, and some lines may be longer than `width`. (Long words will be put on a line by themselves, in order to minimize the amount by which `width` is exceeded.)

**break_on_hyphens**
(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly inseparable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

**max_lines**
(default: `None`) If not `None`, then the output will contain at most `max_lines` lines, with `placeholder` appearing at the end of the
output.

*New in version 3.4.*

**placeholder**

(default: ' [...]'') String that will appear at the end of the output text if it has been truncated.

*New in version 3.4.*

**TextWrapper** also provides some public methods, analogous to the module-level convenience functions:

**wrap**(text)

Wraps the single paragraph in text (a string) so every line is at most width characters long. All wrapping options are taken from instance attributes of the **TextWrapper** instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

**fill**(text)

Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph.
unicodedata — Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 11.0.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, “Unicode Character Database”. It defines the following functions:

**unicodedata.**

**lookup(name)**

Look up character by name. If a character with the given name is found, return the corresponding character. If not found, `KeyError` is raised.

*Changed in version 3.3:* Support for name aliases [1] and named sequences [2] has been added.

**unicodedata.**

**name(chr[, default])**

Returns the name assigned to the character `chr` as a string. If no name is defined, `default` is returned, or, if not given, `ValueError` is raised.

**unicodedata.**

**decimal(chr[, default])**

Returns the decimal value assigned to the character `chr` as integer. If no such value is defined, `default` is returned, or, if not given, `ValueError` is raised.
unicodedata.\texttt{digit}(\texttt{chr}, \texttt{default})

Returns the digit value assigned to the character \texttt{chr} as integer. If no such value is defined, \texttt{default} is returned, or, if not given, \texttt{ValueError} is raised.

unicodedata.\texttt{numeric}(\texttt{chr}, \texttt{default})

Returns the numeric value assigned to the character \texttt{chr} as float. If no such value is defined, \texttt{default} is returned, or, if not given, \texttt{ValueError} is raised.

unicodedata.\texttt{category}(\texttt{chr})

Returns the general category assigned to the character \texttt{chr} as string.

unicodedata.\texttt{bidirectional}(\texttt{chr})

Returns the bidirectional class assigned to the character \texttt{chr} as string. If no such value is defined, an empty string is returned.

unicodedata.\texttt{combining}(\texttt{chr})

Returns the canonical combining class assigned to the character \texttt{chr} as integer. Returns 0 if no combining class is defined.

unicodedata.\texttt{east\_asian\_width}(\texttt{chr})

Returns the east asian width assigned to the character \texttt{chr} as string.

unicodedata.\texttt{mirrored}(\texttt{chr})

Returns the mirrored property assigned to the character \texttt{chr} as integer. Returns 1 if the character has been identified as a “mirrored” character in bidirectional text, 0 otherwise.

unicodedata.\texttt{decomposition}(\texttt{chr})
Returns the character decomposition mapping assigned to the character \texttt{chr} as string. An empty string is returned in case no such mapping is defined.

\texttt{unicodedata.normalize}(\textit{form}, \textit{unistr})

Return the normal form \textit{form} for the Unicode string \textit{unistr}. Valid values for \textit{form} are ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the
compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.

In addition, the module exposes the following constant:

unicodedata.unidata_version

The version of the Unicode database used in this module.

unicodedata.ucd_3_2_0

This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

Examples:

```python
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('٠')  # 'A'rabic, 'AN'
```
Footnotes

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for “equality”. Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of “printable” characters.

**RFC 3454** defines a procedure for “preparing” Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module stringprep only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and
mappings. For a set, `stringprep` provides the “characteristic function”, i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

`stringprep.in_table_a1(code)`
Determine whether `code` is in tableA.1 (Unassigned code points in Unicode 3.2).

`stringprep.in_table_b1(code)`
Determine whether `code` is in tableB.1 (Commonly mapped to nothing).

`stringprep.map_table_b2(code)`
Return the mapped value for `code` according to tableB.2 (Mapping for case-folding used with NFKC).

`stringprep.map_table_b3(code)`
Return the mapped value for `code` according to tableB.3 (Mapping for case-folding used with no normalization).

`stringprep.in_table_c11(code)`
Determine whether `code` is in tableC.1.1 (ASCII space characters).

`stringprep.in_table_c12(code)`
Determine whether `code` is in tableC.1.2 (Non-ASCII space characters).

`stringprep.in_table_c11_c12(code)`
Determine whether `code` is in tableC.1 (Space characters, union of C.1.1 and C.1.2).
stringprep.in_table_c21(code)
  Determine whether code is in tableC.2.1 (ASCII control characters).

stringprep.in_table_c22(code)
  Determine whether code is in tableC.2.2 (Non-ASCII control characters).

stringprep.in_table_c21_c22(code)
  Determine whether code is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.in_table_c3(code)
  Determine whether code is in tableC.3 (Private use).

stringprep.in_table_c4(code)
  Determine whether code is in tableC.4 (Non-character code points).

stringprep.in_table_c5(code)
  Determine whether code is in tableC.5 (Surrogate codes).

stringprep.in_table_c6(code)
  Determine whether code is in tableC.6 (Inappropriate for plain text).

stringprep.in_table_c7(code)
  Determine whether code is in tableC.7 (Inappropriate for canonical representation).

stringprep.in_table_c8(code)
  Determine whether code is in tableC.8 (Change display properties or are deprecated).
stringprep.in_table_c9(code)
Determine whether code is in tableC.9 (Tagging characters).

stringprep.in_table_d1(code)
Determine whether code is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.in_table_d2(code)
Determine whether code is in tableD.2 (Characters with bidirectional property “L”).
The `readline` module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the `rlcompleter` module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in `input()` function.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

**Note:** The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On macOS the `readline` module detects which library is being used at run time.

The configuration file for `libedit` is different from that of GNU readline. If you programmatically load configuration strings you can check for the text “libedit” in `readline.__doc__` to differentiate between GNU readline and `libedit`.

If you use `editline/libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON vi
keybindings and TAB completion:

```python
python:bind -v
python:bind ^I rl_complete
```
Init file

The following functions relate to the init file and user configuration:

`readline.parse_and_bind(string)`
   Execute the init line provided in the `string` argument. This calls `rl_parse_and_bind()` in the underlying library.

`readline.read_init_file([filename])`
   Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library.
Line buffer

The following functions operate on the line buffer:

**readline.** get_line_buffer()

Return the current contents of the line buffer (rl_line_buffer in the underlying library).

**readline.** insert_text(string)

Insert text into the line buffer at the cursor position. This calls rl_insert_text() in the underlying library, but ignores the return value.

**readline.** redisplay()

Change what’s displayed on the screen to reflect the current contents of the line buffer. This calls rl_redisplay() in the underlying library.
History file

The following functions operate on a history file:

readline. `read_history_file([filename])`

Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls `read_history()` in the underlying library.

readline. `write_history_file([filename])`

Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls `write_history()` in the underlying library.

readline. `append_history_file(nelements[, filename])`

Append the last `nelements` items of history to a file. The default filename is `~/.history`. The file must already exist. This calls `append_history()` in the underlying library. This function only exists if Python was compiled for a version of the library that supports it.

    New in version 3.5.

readline. `get_history_length()`
readline. `set_history_length(length)`

Set or return the desired number of lines to save in the history file. The `write_history_file()` function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.
History list

The following functions operate on a global history list:

readline. **clear_history()**

Clear the current history. This calls clear_history() in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

readline. **get_current_history_length()**

Return the number of items currently in the history. (This is different from get_history_length(), which returns the maximum number of lines that will be written to a history file.)

readline. **get_history_item(index)**

Return the current contents of history item at index. The item index is one-based. This calls history_get() in the underlying library.

readline. **remove_history_item(pos)**

Remove history item specified by its position from the history. The position is zero-based. This calls remove_history() in the underlying library.

readline. **replace_history_item(pos, line)**

Replace history item specified by its position with line. The position is zero-based. This calls replace_history_entry() in the underlying library.

readline. **add_history(line)**

Append line to the history buffer, as if it was the last line typed. This calls add_history() in the underlying library.
readline.\texttt{set\_auto\_history}(\texttt{enabled})

Enable or disable automatic calls to \texttt{add\_history()} when reading input via readline. The \texttt{enabled} argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

\textit{New in version 3.6.}

\textbf{CPython implementation detail:} Auto history is enabled by default, and changes to this do not persist across multiple sessions.
Startup hooks

readline.set_startup_hook([function])

Set or remove the function invoked by the rl_startup_hook callback of the underlying library. If function is specified, it will be used as the new hook function; if omitted or None, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.set_pre_input_hook([function])

Set or remove the function invoked by the rl_pre_input_hook callback of the underlying library. If function is specified, it will be used as the new hook function; if omitted or None, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.
Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by `rlcompleter` to complete Python identifiers for the interactive interpreter. If the `readline` module is to be used with a custom completer, a different set of word delimiters should be set.

`readline.set_completer([function])`

Set or remove the completer function. If `function` is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for `state` in `0, 1, 2, …`, until it returns a non-string value. It should return the next possible completion starting with `text`.

The installed completer function is invoked by the `entry_func` callback passed to `rl_completion_matches()` in the underlying library. The `text` string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

`readline.get_completer()`

Get the completer function, or `None` if no completer function has been set.

`readline.get_completion_type()`

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an
integer.

readline.**get_begidx()**
readline.**get_endidx()**

Get the beginning or ending index of the completion scope. These indexes are the start and end arguments passed to the *rl_attempted_completion_function* callback of the underlying library.

readline.**set_completer_delims**(string)
readline.**get_completer_delims()**

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the *rl_completer_word_break_characters* variable in the underlying library.

readline.**set_completion_display_matches_hook**(function)

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or None, any completion display function already installed is removed. This sets or clears the *rl_completion_display_matches_hook* callback in the underlying library. The completion display function is called as 

```c
function(substitution, [matches], longest_match_length)
```

once each time matches need to be displayed.
Example

The following example demonstrates how to use the `readline` module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's `PYTHONSTARTUP` file.

```python
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history"

try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see Readline configuration).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```python
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history"
```
try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len)
atexit.register(save, h_len, histfile)

The following example extends the code.InteractiveConsole class to support history save/restore.

```python
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>", histfile=os.path.expanduser("~/.console-history"), code.InteractiveConsole.__init__(self, locals=None), self.init_history(histfile)):

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
        atexit.register(self.save_history, histfile)
```
```python
def save_history(self, histfile):
    readline.set_history_length(1000)
    readline.write_history_file(histfile)
```

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
rlcompleter — Completion function for GNU readline

Source code: Lib/rlcompleter.py

The rlcompleter module defines a completion function suitable for the readline module by completing valid Python identifiers and keywords.

When this module is imported on a Unix platform with the readline module available, an instance of the Completer class is automatically created and its complete() method is set as the readline completer.

Example:

```python
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__ readline.get_line_buffer(
readline.__file__ readline.insert_text(
readline.__name__ readline.parse_and_bind(
>>> readline.
```

The rlcompleter module is designed for use with Python's interactive mode. Unless Python is run with the -S option, the module is automatically imported and configured (see Readline configuration).

On platforms without readline, the Completer class defined by
this module can still be used for custom purposes.
Completer Objects

Completer objects have the following method:

Completer.complete(text, state)

Return the state\text{th} completion for text.

If called for text that doesn't include a period character (\text{'.'}), it will complete from names currently defined in __main__, builtins and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to \texttt{__getattr__()} up to the last part, and find matches for the rest via the \texttt{dir()} function. Any exception raised during the evaluation of the expression is caught, silenced and \texttt{None} is returned.
Binary Data Services

The modules described in this chapter provide some basic services operations for manipulation of binary data. Other operations on binary data, specifically in relation to file formats and network protocols, are described in the relevant sections.

Some libraries described under Text Processing Services also work with either ASCII-compatible binary formats (for example, `re`) or all binary data (for example, `difflib`).

In addition, see the documentation for Python's built-in binary data types in Binary Sequence Types — bytes, bytearray, memoryview.

- **struct** — Interpret bytes as packed binary data
  - Functions and Exceptions
  - Format Strings
    - Byte Order, Size, and Alignment
    - Format Characters
    - Examples
  - Classes
- **codecs** — Codec registry and base classes
  - Codec Base Classes
    - Error Handlers
    - Stateless Encoding and Decoding
    - Incremental Encoding and Decoding
      - IncrementalEncoder Objects
      - IncrementalDecoder Objects
    - Stream Encoding and Decoding
      - StreamWriter Objects
      - StreamReader Objects
      - StreamReaderWriter Objects
StreamRecoder Objects

- Encodings and Unicode
- Standard Encodings
- Python Specific Encodings
  - Text Encodings
  - Binary Transforms
  - Text Transforms
- encodings.idna — Internationalized Domain Names in Applications
- encodings.mbc — Windows ANSI codepage
- encodings.utf_8_sig — UTF-8 codec with BOM signature
**struct** — Interpret bytes as packed binary data

**Source code:** Lib/struct.py

This module performs conversions between Python values and C structs represented as Python `bytes` objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses Format Strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

**Note:** By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use standard size and alignment instead of native size and alignment: see Byte Order, Size, and Alignment for details.

Several `struct` functions (and methods of `Struct`) take a `buffer` argument. This refers to objects that implement the Buffer Protocol and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.
Functions and Exceptions

The module defines the following exception and functions:

`exception struct.error`
Exception raised on various occasions; argument is a string describing what is wrong.

`struct.pack(format, v1, v2, ...)`
Return a bytes object containing the values `v1`, `v2`, ... packed according to the format string `format`. The arguments must match the values required by the format exactly.

`struct.pack_into(format, buffer, offset, v1, v2, ...)`
Pack the values `v1`, `v2`, ... according to the format string `format` and write the packed bytes into the writable buffer `buffer` starting at position `offset`. Note that `offset` is a required argument.

`struct.unpack(format, buffer)`
Unpack from the buffer `buffer` (presumably packed by `pack(format, ...)`) according to the format string `format`. The result is a tuple even if it contains exactly one item. The buffer’s size in bytes must match the size required by the format, as reflected by `calcsize()`.

`struct.unpack_from(format, buffer, offset=0)`
Unpack from `buffer` starting at position `offset`, according to the format string `format`. The result is a tuple even if it contains exactly one item. The buffer’s size in bytes, minus `offset`, must be at least the size required by the format, as reflected by `calcsize()`.

`struct.iter_unpack(format, buffer)`
Iteratively unpack from the buffer buffer according to the format string format. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer’s size in bytes must be a multiple of the size required by the format, as reflected by calcsize().

Each iteration yields a tuple as specified by the format string.

*New in version 3.4.*

struct.calcsize(format)

Return the size of the struct (and hence of the bytes object produced by pack(format, ...)) corresponding to the format string format.
Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from **Format Characters**, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the **Byte Order, Size, and Alignment**.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Byte order</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>native</td>
<td>native</td>
<td>native</td>
</tr>
<tr>
<td>=</td>
<td>native</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&lt;</td>
<td>little-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&gt;</td>
<td>big-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>!</td>
<td>network (= big-endian)</td>
<td>standard</td>
<td>none</td>
</tr>
</tbody>
</table>

If the first character is not one of these, `@` is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel
Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system.

Native size and alignment are determined using the C compiler’s `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the Format Characters section.

Note the difference between `'@'` and `'=':` both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` is available for those poor souls who claim they can’t remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<` or `'>`.

Notes:

1. Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.
2. No padding is added when using non-native size and alignment, e.g. with `'<`, `'>`, `'=`, and `'!'`.
3. To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See Examples.

**Format Characters**

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The ‘Standard size’ column refers to the size of the packed value in
bytes when using standard size; that is, when the format string starts with one of '<', '>', '!', or '='. When using native size, the size of the packed value is platform-dependent.

<table>
<thead>
<tr>
<th>Format</th>
<th>C Type</th>
<th>Python type</th>
<th>Standard size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>pad byte</td>
<td>no value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>bytes of length 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>signed char</td>
<td>integer</td>
<td>1</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>B</td>
<td>unsigned char</td>
<td>integer</td>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>?</td>
<td>_Bool</td>
<td>bool</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>H</td>
<td>unsigned short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>I</td>
<td>unsigned int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>L</td>
<td>unsigned long</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>q</td>
<td>long long</td>
<td>integer</td>
<td>8</td>
<td>(2),(3)</td>
</tr>
<tr>
<td>Q</td>
<td>unsigned long long</td>
<td>integer</td>
<td>8</td>
<td>(2),(3)</td>
</tr>
<tr>
<td>n</td>
<td>ssize_t</td>
<td>integer</td>
<td></td>
<td>(4)</td>
</tr>
<tr>
<td>N</td>
<td>size_t</td>
<td>integer</td>
<td></td>
<td>(4)</td>
</tr>
<tr>
<td>e</td>
<td>(7)</td>
<td>float</td>
<td>2</td>
<td>(5)</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>float</td>
<td>4</td>
<td>(5)</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>float</td>
<td>8</td>
<td>(5)</td>
</tr>
<tr>
<td>s</td>
<td>char[]</td>
<td>bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>--------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>char[]</td>
<td>bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>void *</td>
<td>integer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Changed in version 3.3:* Added support for the 'n' and 'N' formats.

*Changed in version 3.6:* Added support for the 'e' format.

**Notes:**

1. The '?' conversion code corresponds to the _Bool type defined by C99. If this type is not available, it is simulated using a char. In standard mode, it is always represented by one byte.

2. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

3. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a __index__() method then that method is called to convert the argument to an integer before packing.

   *Changed in version 3.2:* Use of the __index__() method for non-integers is new in 3.2.

4. The 'n' and 'N' conversion codes are only available for the native size (selected as the default or with the '@' byte order character). For the standard size, you can use whichever of the other integer formats fits your application.
5. For the 'f', 'd' and 'e' conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for 'f', 'd' or 'e' respectively), regardless of the floating-point format used by the platform.

6. The 'P' format character is only available for the native byte ordering (selected as the default or with the '@' byte order character). The byte order character '=' chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the 'P' format is not available.

7. The IEEE 754 binary16 “half precision” type was introduced in the 2008 revision of the IEEE 754 standard. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately 6.1e-05 and 6.5e+04 at full precision. This type is not widely supported by C compilers: on a typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format for more information.

A format character may be preceded by an integral repeat count. For example, the format string '4h' means exactly the same as 'hhhh'.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the 's' format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it
For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0 characters).

When packing a value \( x \) using one of the integer formats ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q'), if \( x \) is outside the valid range for that format then `struct.error` is raised.

*Changed in version 3.1:* In 3.0, some of the integer formats wrapped out-of-range values and raised `DeprecationWarning` instead of `struct.error`.

The 'p' format character encodes a “Pascal string”, meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to `pack()` is too long (longer than the count minus 1), only the leading `count-1` bytes of the string are stored. If the string is shorter than `count-1`, it is padded with null bytes so that exactly `count` bytes in all are used. Note that for `unpack()`, the 'p' format character consumes `count` bytes, but that the string returned can never contain more than 255 bytes.

For the '?' format character, the return value is either `True` or `False`. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be `True` when unpacking.

**Examples**

**Note:** All examples assume a native byte order, size, and alignment with a big-endian machine.
A basic example of packing/unpacking three integers:

```python
gesp der struc tirp *
gesp('hh1', 1, 2, 3) b'\x00\x01\x00\x02\x00\x00\x00\x03'
gesp('hh1', b'\x00\x01\x00\x02\x00\x00\x00\x03', (1, 2, 3)
gespsize('hh1') 8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```python
gesp record = b'raymond \x32\x12\x08\x01\x08'
gesp name, serialnum, school, gradelevel = gesp('<10sHHb', record)
gesp from collections tirp namedtuple
gesp Student = namedtuple('Student', 'name serialnum school gradelevel')
gesp Student._make(gesp('<10sHHb', record))
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```python
gesp pack('ci', b'*', 0x12131415) b'*\x00\x00\x00\x12\x13\x14\x15'
gesp pack('ic', 0x12131415, b'*') b'\x12\x13\x14\x15'
gesp calcsiz('ci') 8
gesp calcsiz('ic') 5
The following format 'llh0l' specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```python
>>> pack('llh0l', 1, 2, 3)
'b'\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

**See also:**

**Module array**
- Packed binary storage of homogeneous data.

**Module xdrlib**
- Packing and unpacking of XDR data.
Classes

The `struct` module also defines the following type:

```python
class struct.Struct(format)
```

Return a new Struct object which writes and reads binary data according to the format string `format`. Creating a Struct object once and calling its methods is more efficient than calling the `struct` functions with the same format since the format string only needs to be compiled once.

**Note:** The compiled versions of the most recent format strings passed to `Struct` and the module-level functions are cached, so programs that use only a few format strings needn’t worry about reusing a single `Struct` instance.

Compiled Struct objects support the following methods and attributes:

**pack**(*v1, v2, ...*)

Identical to the `pack()` function, using the compiled format. *(len(result) will equal `size`.)*

**pack_into**(*buffer, offset, v1, v2, ...*)

Identical to the `pack_into()` function, using the compiled format.

**unpack**(*buffer*)

Identical to the `unpack()` function, using the compiled format. The buffer’s size in bytes must equal `size`.

**unpack_from**(*buffer, offset=0*)
Identical to the `unpack_from()` function, using the compiled format. The buffer’s size in bytes, minus \textit{offset}, must be at least \textit{size}.

\textbf{iter\_unpack}(\textit{buffer})

Identical to the `iter\_unpack()` function, using the compiled format. The buffer’s size in bytes must be a multiple of \textit{size}.

\textit{New in version 3.4.}

\textbf{format}

The format string used to construct this Struct object.

\textit{Changed in version 3.7:} The format string type is now \texttt{str} instead of \texttt{bytes}.

\textbf{size}

The calculated size of the struct (and hence of the bytes object produced by the \texttt{pack()} method) corresponding to \texttt{format}.
**codecs — Codec registry and base classes**

**Source code:** Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are **text encodings**, which encode text to bytes, but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to use specifically with **text encodings**, or with codecs that encode to **bytes**.

The module defines the following functions for encoding and decoding with any codec:

```python
import codecs

codecs.encode(obj, encoding='utf-8', errors='strict')

Encodes `obj` using the codec registered for `encoding`.

*Errors* may be given to set the desired error handling scheme. The default error handler is `strict` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to **Codec Base Classes** for more information on codec error handling.

```python
codecs.decode(obj, encoding='utf-8', errors='strict')

Decodes `obj` using the codec registered for `encoding`.

*Errors* may be given to set the desired error handling scheme. The
The default error handler is 'strict' meaning that decoding errors raise ValueError (or a more codec specific subclass, such as UnicodeDecodeError). Refer to Codec Base Classes for more information on codec error handling.

The full details for each codec can also be looked up directly:

```python
codecs lookup(encoding)
```
Looks up the codec info in the Python codec registry and returns a CodecInfo object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no CodecInfo object is found, a LookupError is raised. Otherwise, the CodecInfo object is stored in the cache and returned to the caller.

```python
class codecsCodecInfo(encode, decode, streamreader=None, streamwriter=None, incrementalencoder=None, incrementaldecoder=None, name=None)
```
Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

**name**
The name of the encoding.

**encode**
**decode**
The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the encode() and decode() methods of Codec instances (see Codec Interface). The functions or methods are expected to work in a stateless mode.
incrementalencoder
incrementaldecoder

Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes \texttt{IncrementalEncoder} and \texttt{IncrementalDecoder}, respectively. Incremental codecs can maintain state.

streamwriter
streamreader

Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes \texttt{StreamWriter} and \texttt{StreamReader}, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use \texttt{lookup()} for the codec lookup:

\begin{verbatim}

codecs.getencoder(encoding)

Look up the codec for the given encoding and return its encoder function.

Raises a \texttt{LookupError} in case the encoding cannot be found.

\end{verbatim}

\begin{verbatim}

codecs.getdecoder(encoding)

Look up the codec for the given encoding and return its decoder function.

Raises a \texttt{LookupError} in case the encoding cannot be found.

\end{verbatim}

\begin{verbatim}

codecs.getincrementalencoder(encoding)

Look up the codec for the given encoding and return its incremental encoder class or factory function.

\end{verbatim}
Raises a LookupError in case the encoding cannot be found or
the codec doesn’t support an incremental encoder.

```
codecs.getincrementaldecoder(encoding)
```
Look up the codec for the given encoding and return its
incremental decoder class or factory function.

Raises a LookupError in case the encoding cannot be found or
the codec doesn’t support an incremental decoder.

```
codecs.getreader(encoding)
```
Look up the codec for the given encoding and return its
StreamReader class or factory function.

Raises a LookupError in case the encoding cannot be found.

```
codecs.getWriter(encoding)
```
Look up the codec for the given encoding and return its
StreamWriter class or factory function.

Raises a LookupError in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec
search function:

```
codecs.register(search_function)
```
Register a codec search function. Search functions are expected to
take one argument, being the encoding name in all lower case
letters, and return a CodecInfo object. In case a search function
cannot find a given encoding, it should return None.

**Note:** Search function registration is not currently reversible,
which may cause problems in some cases, such as unit testing or
While the builtin `open()` and the associated `io` module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

```python
codecs.open(filename, mode='r', encoding=None, errors='strict', buffering=1)
```

Open an encoded file using the given `mode` and return an instance of `StreamReaderWriter`, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

**Note:** Underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The `mode` argument may be any binary mode acceptable to the built-in `open()` function; the `'b'` is automatically added.

`encoding` specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

`errors` may be given to define the error handling. It defaults to `'strict'` which causes a `ValueError` to be raised in case an encoding error occurs.

`buffering` has the same meaning as for the built-in `open()` function. It defaults to line buffered.

```python
codecs.EncodedFile(file, data_encoding, file_encoding=None, errors='strict')
```
Return a `StreamRecoder` instance, a wrapped version of `file` which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given `data_encoding` and then written to the original file as bytes using `file_encoding`. Bytes read from the original file are decoded according to `file_encoding`, and the result is encoded using `data_encoding`.

If `file_encoding` is not given, it defaults to `data_encoding`.

`errors` may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

codecs.\texttt{iterencode}(\textit{iterator, encoding, errors='strict', **kwargs})

Uses an incremental encoder to iteratively encode the input provided by `iterator`. This function is a generator. The `errors` argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text `str` objects to encode. Therefore it does not support bytes-to-bytes encoders such as base64_codec.

codecs.\texttt{iterdecode}(\textit{iterator, encoding, errors='strict', **kwargs})

Uses an incremental decoder to iteratively decode the input provided by `iterator`. This function is a generator. The `errors` argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept `bytes` objects to
decode. Therefore it does not support text-to-text encoders such as rot_13, although rot_13 may be used equivalently with iterencode().

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE

These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. BOM_UTF16 is either BOM_UTF16_BE or BOM_UTF16_LE depending on the platform's native byte order, BOM is an alias for BOM_UTF16, BOM_LE for BOM_UTF16_LE and BOM_BE for BOM_UTF16_BE. The others represent the BOM in UTF-8 and UTF-32 encodings.
Codec Base Classes

The codecs module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the errors string argument. The following string values are defined and implemented by all standard Python codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'strict'</td>
<td>Raise UnicodeError (or a subclass); this is the default. Implemented in strict_errors().</td>
</tr>
<tr>
<td>'ignore'</td>
<td>Ignore the malformed data and continue without further notice. Implemented in ignore_errors().</td>
</tr>
</tbody>
</table>

The following error handlers are only applicable to text encodings:
'replace' | Replace with a suitable replacement marker; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in codecs on decoding, and ‘?’ on encoding. Implemented in `replace_errors()`.

'xmlcharrefreplace' | Replace with the appropriate XML character reference (only for encoding). Implemented in `xmlcharrefreplace_errors()`.

'backslashreplace' | Replace with backslashed escape sequences. Implemented in `backslashreplace_errors()`.

'namereplace' | Replace with \N{...} escape sequences (only for encoding). Implemented in `namereplace_errors()`.

'surrogateescape' | On decoding, replace byte with individual surrogate code ranging from U+DC80 to U+DCFF. This code will then be turned back into the same byte when the 'surrogateescape' error handler is used when encoding the data. (See PEP 383 for more.)

In addition, the following error handler is specific to the given codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Codecs</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'surrogatepass'</td>
<td>utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le</td>
<td>Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.</td>
</tr>
</tbody>
</table>
New in version 3.1: The 'surrogateescape' and 'surrogatepass' error handlers.

Changed in version 3.4: The 'surrogatepass' error handlers now works with utf-16* and utf-32* codecs.

New in version 3.5: The 'namereplace' error handler.

Changed in version 3.5: The 'backslashreplace' error handlers now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

codecs.register_error(name, error_handler)

Register the error handling function error_handler under the name name. The error_handler argument will be called during encoding and decoding in case of an error, when name is specified as the errors parameter.

For encoding, error_handler will be called with a UnicodeEncodeError instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either str or bytes. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an IndexError will be raised.
Decoding and translating works similarly, except `UnicodeDecodeError` or `UnicodeTranslateError` will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

```
codecs.lookup_error(name)
```

Return the error handler previously registered under the name `name`.

Raises a `LookupError` in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

```
codecs.strict_errors(exception)
```

Implements the `'strict' error handling: each encoding or decoding error raises a `UnicodeError`.

```
codecs.replace_errors(exception)
```

Implements the `'replace' error handling (for text encodings only): substitutes '?' for encoding errors (to be encoded by the codec), and `�` (the Unicode replacement character) for decoding errors.

```
codecs.ignore_errors(exception)
```

Implements the `'ignore' error handling: malformed data is ignored and encoding or decoding is continued without further notice.

```
codecs.xmlcharrefreplace_errors(exception)
```
Implements the 'xmlcharrefreplace' error handling (for encoding with text encodings only): the unencodable character is replaced by an appropriate XML character reference.

codecs.\texttt{backslashreplace\_errors}(\texttt{exception})
Implements the 'backslashreplace' error handling (for text encodings only): malformed data is replaced by a backslashed escape sequence.

codecs.\texttt{namereplace\_errors}(\texttt{exception})
Implements the 'namereplace' error handling (for encoding with text encodings only): the unencodable character is replaced by a \texttt{\textbackslash N\{...\}} escape sequence.

\textit{New in version 3.5.}

Stateless Encoding and Decoding

The base Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

\texttt{Codec.\texttt{encode}(input[, errors])}
Encodes the object \texttt{input} and returns a tuple (output object, length consumed). For instance, text encoding converts a string object to a bytes object using a particular character set encoding (e.g., \texttt{cp1252} or \texttt{iso-8859-1}).

The \texttt{errors} argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use \texttt{StreamWriter} for codecs which have to keep state in order to
make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

```
Codec.decode(input[, errors])
```

Decodes the object `input` and returns a tuple (output object, length consumed). For instance, for a text encoding, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, `input` must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The `errors` argument defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use `StreamReader` for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

**Incremental Encoding and Decoding**

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn’t done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the
encoding/decoding process during method calls.

The joined output of calls to the `encode() / decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

```python
class codecs.IncrementalEncoder(errors='strict')
    Constructor for an IncrementalEncoder instance.
```

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the `errors` keyword argument. See Error Handlers for possible values.

The `errors` argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

```python
encode(object[, final])
    Encodes `object` (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` `final` must be true (the default is false).
```
reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that 0 is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer).

setstate(state)

Set the state of the encoder to state. state must be an encoder state returned by getstate().

IncrementalDecoder Objects

The IncrementalDecoder class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.IncrementalDecoder(errors='strict')

Constructor for an IncrementalDecoder instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.
The **IncrementalDecoder** may implement different error handling schemes by providing the *errors* keyword argument. See **Error Handlers** for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the **IncrementalDecoder** object.

### decode(*object[, final]*)
Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to `decode()` *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn’t possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

### reset()
Reset the decoder to the initial state.

### getstate()
Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that 0 is the most common additional state info.) If this additional state info is 0 it must be possible to set the decoder to the state which has no input buffered and 0 as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any
output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

\texttt{setstate(state)}

Set the state of the encoder to \textit{state}. \textit{state} must be a decoder state returned by \texttt{getstate()}. 

Stream Encoding and Decoding

The \texttt{StreamWriter} and \texttt{StreamReader} classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See \texttt{encodings.utf_8} for an example of how this is done.

StreamWriter Objects

The \texttt{StreamWriter} class is a subclass of \texttt{Codec} and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

\texttt{class codecs.StreamWriter(stream, errors='strict')}  
Constructor for a \texttt{StreamWriter} instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The \textit{stream} argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.
The **StreamWriter** may implement different error handling schemes by providing the *errors* keyword argument. See **Error Handlers** for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the **StreamWriter** object.

**write**(object)

  Writes the object’s contents encoded to the stream.

**writelines**(list)

  Writes the concatenated list of strings to the stream (possibly by reusing the **write**() method). The standard bytes-to-bytes codecs do not support this method.

**reset**()

  Flushes and resets the codec buffers used for keeping state.

  
  Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the **StreamWriter** must also inherit all other methods and attributes from the underlying stream.

**StreamReader Objects**

The **StreamReader** class is a subclass of Codec and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.
class codecs StreamReader(stream, errors='strict')

Constructor for a StreamReader instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The stream argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The StreamReader may implement different error handling schemes by providing the errors keyword argument. See Error Handlers for the standard error handlers the underlying stream codec may support.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the StreamReader object.

The set of allowed values for the errors argument can be extended with register_error().

read([size[, chars[, firstline]]])

Decodes data from the stream and returns the resulting object.

The chars argument indicates the number of decoded code points or bytes to return. The read() method will never return more data than requested, but it might return less, if there is not enough available.

The size argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The
decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

**readline**([*size*, *keepends]*)
Read one line from the input stream and return the decoded data.

*size*, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

**readlines**([*sizehint*, *keepends]*)
Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's decoder method and are included in the list entries if *keepends* is true.

*sizehint*, if given, is passed as the *size* argument to the stream's `read()` method.
reset()

Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

```python
class codecs.StreamReaderWriter(stream, Reader, Writer, errors='strict')
```

Creates a `StreamReaderWriter` instance. `stream` must be a file-like object. `Reader` and `Writer` must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another,
which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

```python
class codecs.StreamRecoder(stream, encode, decode, Reader, Writer, errors='strict')
```

Creates a `StreamRecoder` instance which implements a two-way conversion: `encode` and `decode` work on the frontend — the data visible to code calling `read()` and `write()`, while `Reader` and `Writer` work on the backend — the data in `stream`.

You can use these objects to do transparent transcodings from e.g. Latin-1 to UTF-8 and back.

The `stream` argument must be a file-like object.

The `encode` and `decode` arguments must adhere to the Codec interface. `Reader` and `Writer` must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

`StreamRecoder` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.
Strings are stored internally as sequences of code points in range 0x0–0x10FFFF. (See PEP 393 for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as encoding, and recreating the string from the sequence of bytes is known as decoding. There are a variety of different text serialisation codecs, which are collectively referred to as text encodings.

The simplest text encoding (called 'latin-1' or 'iso-8859-1') maps the code points 0–255 to the bytes 0x0–0xff, which means that a string object that contains code points above U+00FF can't be encoded with this codec. Doing so will raise a UnicodeEncodeError that looks like the following (although the details of the error message may differ): UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).

There's another group of encodings (the so-called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes 0x0–0xff. To see how this is done simply open e.g. encodings/cp1252.py (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can
store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there’s the so called BOM (“Byte Order Mark”). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it’s a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it’s a normal character that will be decoded like any other.

There’s another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant
bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

<table>
<thead>
<tr>
<th>Range</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-000000000 ... U-00000007F</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>U-000000080 ... U-0000007FF</td>
<td>110xxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-000000800 ... U-0000FFFFF</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-000100000 ... U-0010FFFFF</td>
<td>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded string (even if it's the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it's rather improbable that any charmap encoded file starts
with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.
Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. 'utf-8' is a valid alias for the 'utf_8' codec.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbc (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Changed in version 3.6: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>646, us-ascii</td>
<td>English</td>
</tr>
<tr>
<td>big5</td>
<td>big5-tw, csbig5</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>big5hkscs</td>
<td>big5-hkscs, hkscs</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp037</td>
<td>IBM037, IBM039</td>
<td>English</td>
</tr>
<tr>
<td>cp273</td>
<td>273, IBM273, csIBM273</td>
<td>German</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>New in version 3.4.</em></td>
</tr>
<tr>
<td>cp424</td>
<td>EBCDIC-CP-HE, IBM424</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp437</td>
<td>437, IBM437</td>
<td>English</td>
</tr>
<tr>
<td>cp500</td>
<td>EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp720</td>
<td></td>
<td>Arabic</td>
</tr>
<tr>
<td>cp737</td>
<td></td>
<td>Greek</td>
</tr>
<tr>
<td>cp775</td>
<td>IBM775</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp850</td>
<td>850, IBM850</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp852</td>
<td>852, IBM852</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>cp855</td>
<td>855, IBM855</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>cp856</td>
<td></td>
<td>Hebrew</td>
</tr>
<tr>
<td>CP</td>
<td>Description</td>
<td>Language</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>cp857</td>
<td>857, IBM857</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp858</td>
<td>858, IBM858</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp860</td>
<td>860, IBM860</td>
<td>Portuguese</td>
</tr>
<tr>
<td>cp861</td>
<td>861, CP-IS, IBM861</td>
<td>Icelandic</td>
</tr>
<tr>
<td>cp862</td>
<td>862, IBM862</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp863</td>
<td>863, IBM863</td>
<td>Canadian</td>
</tr>
<tr>
<td>cp864</td>
<td>IBM864</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp865</td>
<td>865, IBM865</td>
<td>Danish, Norwegian</td>
</tr>
<tr>
<td>cp866</td>
<td>866, IBM866</td>
<td>Russian</td>
</tr>
<tr>
<td>cp869</td>
<td>869, CP-GR, IBM869</td>
<td>Greek</td>
</tr>
<tr>
<td>cp874</td>
<td></td>
<td>Thai</td>
</tr>
<tr>
<td>cp875</td>
<td></td>
<td>Greek</td>
</tr>
<tr>
<td>cp932</td>
<td>932, ms932, mskanji, ms-kanji</td>
<td>Japanese</td>
</tr>
<tr>
<td>cp949</td>
<td>949, ms949, uhc</td>
<td>Korean</td>
</tr>
<tr>
<td>cp950</td>
<td>950, ms950</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp1006</td>
<td></td>
<td>Urdu</td>
</tr>
<tr>
<td>cp1026</td>
<td>ibm1026</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1125</td>
<td>1125, ibm1125, cp866u, ruscii</td>
<td>Ukrainian</td>
</tr>
<tr>
<td>cp1140</td>
<td>ibm1140</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp1250</td>
<td>windows-1250</td>
<td>Central and Eastern Europe</td>
</tr>
</tbody>
</table>

*New in version 3.4.*
<table>
<thead>
<tr>
<th>Code</th>
<th>Windows Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp1251</td>
<td>windows-1251</td>
<td>Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>cp1252</td>
<td>windows-1252</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp1253</td>
<td>windows-1253</td>
<td>Greek</td>
</tr>
<tr>
<td>cp1254</td>
<td>windows-1254</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1255</td>
<td>windows-1255</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp1256</td>
<td>windows-1256</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp1257</td>
<td>windows-1257</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp1258</td>
<td>windows-1258</td>
<td>Vietnamese</td>
</tr>
<tr>
<td>cp65001</td>
<td></td>
<td>Windows only: Windows UTF-8 (CP_UTF8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New in version 3.3.</td>
</tr>
<tr>
<td>euc_jp</td>
<td>eucjp, ujis, u-jis</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_jis_2004</td>
<td>jisx0213, eucjis2004</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_jisx0213</td>
<td>eucjisx0213</td>
<td>Japanese</td>
</tr>
<tr>
<td>euc_kr</td>
<td>euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001</td>
<td>Korean</td>
</tr>
<tr>
<td>gb2312</td>
<td>chinese, csiso58gb231280, euc-cn, eucnn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>gbk</td>
<td>936, cp936, ms936</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>Code</td>
<td>Representation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>gb18030</td>
<td>gb18030-2000</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>hz</td>
<td>hzgb, hz-gb, hz-gb-2312</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>iso2022_jp</td>
<td>csiso2022jp, iso2022jp, iso-2022-jp</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_1</td>
<td>iso2022jp-1, iso-2022-jp-1</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_2</td>
<td>iso2022jp-2, iso-2022-jp-2</td>
<td>Japanese, Korean, Simplified Chinese, Western Europe, Greek</td>
</tr>
<tr>
<td>iso2022_jp_3</td>
<td>iso2022jp-3, iso-2022-jp-3</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_ext</td>
<td>iso2022jp-ext, iso-2022-jp-ext</td>
<td>Japanese</td>
</tr>
<tr>
<td>latin_1</td>
<td>iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1</td>
<td>West Europe</td>
</tr>
<tr>
<td>iso8859_2</td>
<td>iso-8859-2, latin2, L2</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>iso8859_3</td>
<td>iso-8859-3, latin3, L3</td>
<td>Esperanto, Maltese</td>
</tr>
<tr>
<td>iso8859_4</td>
<td>iso-8859-4, latin4, L4</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_5</td>
<td>iso-8859-5, cyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Language</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>iso8859_6</td>
<td>iso-8859-6, arabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>iso8859_7</td>
<td>iso-8859-7, greek, greek8</td>
<td>Greek</td>
</tr>
<tr>
<td>iso8859_8</td>
<td>iso-8859-8, hebrew</td>
<td>Hebrew</td>
</tr>
<tr>
<td>iso8859_9</td>
<td>iso-8859-9, latin5, L5</td>
<td>Turkish</td>
</tr>
<tr>
<td>iso8859_10</td>
<td>iso-8859-10, latin6, L6</td>
<td>Nordic languages</td>
</tr>
<tr>
<td>iso8859_11</td>
<td>iso-8859-11, thai</td>
<td>Thai languages</td>
</tr>
<tr>
<td>iso8859_13</td>
<td>iso-8859-13, latin7, L7</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_14</td>
<td>iso-8859-14, latin8, L8</td>
<td>Celtic languages</td>
</tr>
<tr>
<td>iso8859_15</td>
<td>iso-8859-15, latin9, L9</td>
<td>Western Europe</td>
</tr>
<tr>
<td>iso8859_16</td>
<td>iso-8859-16, latin10, L10</td>
<td>South-Eastern Europe</td>
</tr>
<tr>
<td>johab</td>
<td>cp1361, ms1361</td>
<td>Korean</td>
</tr>
<tr>
<td>koi8_r</td>
<td></td>
<td>Russian</td>
</tr>
<tr>
<td>koi8_t</td>
<td></td>
<td>Tajik</td>
</tr>
<tr>
<td>koi8_u</td>
<td></td>
<td>Ukrainian</td>
</tr>
<tr>
<td>kz1048</td>
<td>kz_1048, strk1048_2002, rk1048</td>
<td>Kazakh</td>
</tr>
<tr>
<td>mac_cyrillic</td>
<td>maccyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>mac_greek</td>
<td>macgreek</td>
<td>Greek</td>
</tr>
<tr>
<td>Encoding</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>mac_iceland</td>
<td>Icelandic</td>
<td></td>
</tr>
<tr>
<td>mac_latin2</td>
<td>maclatin2, maccentraleurope Central and Eastern Europe</td>
<td></td>
</tr>
<tr>
<td>mac_roman</td>
<td>macroman, macintosh Western Europe</td>
<td></td>
</tr>
<tr>
<td>mac_turkish</td>
<td>macturkish Turkish</td>
<td></td>
</tr>
<tr>
<td>ptcp154</td>
<td>csptcp154, pt154, cp154, cyrillic-asian Kazakh</td>
<td></td>
</tr>
<tr>
<td>shift_jis</td>
<td>csshiftjis, shiftjis, sjis, s_jis Japanese</td>
<td></td>
</tr>
<tr>
<td>shift_jisx0213</td>
<td>shiftjisx0213, sjisx0213, s_jisx0213 Japanese</td>
<td></td>
</tr>
<tr>
<td>utf_32</td>
<td>U32, utf32 all languages</td>
<td></td>
</tr>
<tr>
<td>utf_32_be</td>
<td>UTF-32BE all languages</td>
<td></td>
</tr>
<tr>
<td>utf_32_le</td>
<td>UTF-32LE all languages</td>
<td></td>
</tr>
<tr>
<td>utf_16</td>
<td>U16, utf16 all languages</td>
<td></td>
</tr>
<tr>
<td>utf_16_be</td>
<td>UTF-16BE all languages</td>
<td></td>
</tr>
<tr>
<td>utf_16_le</td>
<td>UTF-16LE all languages</td>
<td></td>
</tr>
<tr>
<td>utf_7</td>
<td>U7, unicode-1-1-utf-7 all languages</td>
<td></td>
</tr>
<tr>
<td>utf_8</td>
<td>U8, UTF, utf8 all languages</td>
<td></td>
</tr>
<tr>
<td>utf_8_sig</td>
<td>all languages</td>
<td></td>
</tr>
</tbody>
</table>

*Changed in version 3.4:* The utf-16* and utf-32* encoders no longer allow surrogate code points (U+D800–U+DFFF) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.
Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

Text Encodings

The following codecs provide *str* to *bytes* encoding and *bytes-like object* to *str* decoding, similar to the Unicode text encodings.

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>idna</td>
<td></td>
<td>Implements <a href="https://tools.ietf.org/html/rfc3490">RFC 3490</a>, see also <em>encodings.idna</em>. Only <code>errors='strict'</code> is supported.</td>
</tr>
<tr>
<td>mbcscs</td>
<td>ansi, dbcs</td>
<td>Windows only: Encode operand according to the ANSI codepage (CP_ACP)</td>
</tr>
</tbody>
</table>
| oem    |         | Windows only: Encode operand according to the OEM codepage (CP_OEMCP)  

*New in version 3.6.*

<p>| palmos |         | Encoding of PalmOS 3.5 |</p>
<table>
<thead>
<tr>
<th><strong>punycode</strong></th>
<th>Implements RFC 3492. Stateful codecs are not supported.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>raw_unicode_escape</strong></td>
<td>Latin-1 encoding with \uXXXX and \UXXXXXXXX for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.</td>
</tr>
<tr>
<td><strong>undefined</strong></td>
<td>Raise an exception for all conversions, even empty strings. The error handler is ignored.</td>
</tr>
<tr>
<td><strong>unicode_escape</strong></td>
<td>Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decodes from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.</td>
</tr>
</tbody>
</table>
| **unicode_internal** | Return the internal representation of the operand. Stateful codecs are not supported.  

*Deprecated since version 3.3: This representation is obsoleted by PEP 393.* |

**Binary Transforms**

The following codecs provide binary transforms: *bytes-like object* to
bytes mappings. They are not supported by `bytes.decode()` (which only produces str output).

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
<th>Encoder / decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64_codec</td>
<td>base64, base_64</td>
<td>Convert operand to multiline MIME base64 (the result always includes a trailing \n)</td>
<td>base64.encod / base64.decode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Changed in version 3.4: accepts any bytes-like object as input for encoding and decoding</td>
<td></td>
</tr>
<tr>
<td>bz2_codec</td>
<td>bz2</td>
<td>Compress the operand using bz2</td>
<td>bz2.compress / bz2.decompress</td>
</tr>
<tr>
<td>hex_codec</td>
<td>hex</td>
<td>Convert operand to hexadecimal representation, with two digits per byte</td>
<td>binascii.b2a / binascii.a2b</td>
</tr>
<tr>
<td>quopri_codec</td>
<td>quopri, quotedprintable, quoted_printable</td>
<td>Convert operand to MIME quoted</td>
<td>quopri.encod quotetabs=True / quopri.decod</td>
</tr>
<tr>
<td>Codec</td>
<td>Aliases</td>
<td>Purpose</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>uu_codec</td>
<td>uu</td>
<td>Convert the operand using uuencode</td>
<td></td>
</tr>
<tr>
<td>zlib_codec</td>
<td>zip, zlib</td>
<td>Compress the operand using gzip</td>
<td></td>
</tr>
</tbody>
</table>

In addition to bytes-like objects, `base64_codec` also accepts ASCII-only instances of `str` for decoding.

*New in version 3.2:* Restoration of the binary transforms.

*Changed in version 3.4:* Restoration of the aliases for the binary transforms.

## Text Transforms

The following codec provides a text transform: a `str` to `str` mapping. It is not supported by `str.encode()` (which only produces `bytes` output).

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rot_13</td>
<td>rot13</td>
<td>Returns the Caesar-cypher encryption of the operand</td>
</tr>
</tbody>
</table>

*New in version 3.2:* Restoration of the `rot_13` text transform.

*Changed in version 3.4:* Restoration of the `rot13` alias.
**encodings.idna** — Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and **stringprep**.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP Host fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the separator and converting any ACE labels found into unicode. Furthermore, the `socket` module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function
parameters, such as `http.client` and `ftplib`, accept Unicode host names (`http.client` then also transparently sends an IDNA hostname in the `Host` field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module `encodings.idna` also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

```
encodings.idna.nameprep(label)
```

Return the nameprepped version of `label`. The implementation currently assumes query strings, so `AllowUnassigned` is true.

```
encodings.idna.ToASCII(label)
```

Convert a label to ASCII, as specified in `RFC 3490`. `UseSTD3ASCIIRules` is assumed to be false.

```
encodings.idna.ToUnicode(label)
```

Convert a label to Unicode, as specified in `RFC 3490`. 
**encodings.mbc** — Windows ANSI codepage

Encode operand according to the ANSI codepage (CP_ACP).

**Availability**: Windows only.

*Changed in version 3.3*: Support any error handler.

*Changed in version 3.2*: Before 3.2, the `errors` argument was ignored; `replace` was always used to encode, and `ignore` to decode.
encodings.utf_8_sig — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.
Data Types

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, `dict`, `list`, `set` and `frozenset`, and `tuple`. The `str` class is used to hold Unicode strings, and the `bytes` class is used to hold binary data.

The following modules are documented in this chapter:

- **datetime** — Basic date and time types
  - Available Types
  - `timedelta` Objects
  - `date` Objects
  - `datetime` Objects
  - `time` Objects
  - `tzinfo` Objects
  - `timezone` Objects
  - `strftime()` and `strptime()` Behavior
- **calendar** — General calendar-related functions
- **collections** — Container datatypes
  - `ChainMap` objects
    - `ChainMap` Examples and Recipes
  - `Counter` objects
  - `deque` objects
    - `deque` Recipes
  - `defaultdict` objects
    - `defaultdict` Examples
  - `namedtuple()` Factory Function for Tuples with Named
Fields
  - OrderedDict objects
    - OrderedDict Examples and Recipes
  - UserDict objects
  - UserList objects
  - UserString objects

- collections.abc — Abstract Base Classes for Containers
  - Collections Abstract Base Classes

- heapq — Heap queue algorithm
  - Basic Examples
  - Priority Queue Implementation Notes
  - Theory

- bisect — Array bisection algorithm
  - Searching Sorted Lists
  - Other Examples

- array — Efficient arrays of numeric values

- weakref — Weak references
  - Weak Reference Objects
  - Example
  - Finalizer Objects
  - Comparing finalizers with __del__() methods

- types — Dynamic type creation and names for built-in types
  - Dynamic Type Creation
  - Standard Interpreter Types
  - Additional Utility Classes and Functions
  - Coroutine Utility Functions

- copy — Shallow and deep copy operations

- pprint — Data pretty printer
  - PrettyPrinter Objects
  - Example

- reprlib — Alternate repr() implementation
  - Repr Objects
  - Subclassing Repr Objects
• enum — Support for enumerations
  • Module Contents
  • Creating an Enum
  • Programmatic access to enumeration members and their attributes
  • Duplicating enum members and values
  • Ensuring unique enumeration values
  • Using automatic values
  • Iteration
  • Comparisons
  • Allowed members and attributes of enumerations
  • Restricted Enum subclassing
  • Pickling
  • Functional API
  • Derived Enumerations
    • IntEnum
    • IntFlag
    • Flag
    • Others
  • Interesting examples
    • Omitting values
      • Using auto
      • Using object
      • Using a descriptive string
      • Using a custom __new__( )
    • OrderedEnum
    • DuplicateFreeEnum
    • Planet
    • TimePeriod
  • How are Enums different?
    • Enum Classes
    • Enum Members (aka instances)
    • Finer Points
      • Supported __dunder__ names
- Supported _sunder_ names
- Enum member type
- Boolean value of Enum classes and members
- Enum classes with methods
- Combining members of Flag
The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the `time` and `calendar` modules.

There are two kinds of date and time objects: “naive” and “aware”.

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation [1].

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, `datetime` and `time` objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract `tzinfo` class. These
tzinfo objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that only one concrete tzinfo class, the timezone class, is supplied by the datetime module. The timezone class can represent simple timezones with fixed offset from UTC, such as UTC itself or North American EST and EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

The datetime module exports the following constants:

datetime.MINYEAR
The smallest year number allowed in a date or datetime object. MINYEAR is 1.

datetime.MAXYEAR
The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

See also:

Module calendar
General calendar related functions.

Module time
Time access and conversions.
Available Types

**class datetime.date**
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

**class datetime.time**
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of “leap seconds” here). Attributes: hour, minute, second, microsecond, and tzinfo.

**class datetime.datetime**
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

**class datetime.timedelta**
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

**class datetime.tzinfo**
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

**class datetime.timezone**
A class that implements the tzinfo abstract base class as a fixed offset from the UTC.
New in version 3.2.

Objects of these types are immutable.

Objects of the `date` type are always naive.

An object of type `time` or `datetime` may be naive or aware. A `datetime` object `d` is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, `d` is naive. A `time` object `t` is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, `t` is naive.

The distinction between naive and aware doesn't apply to `timedelta` objects.

Subclass relationships:
timedelta Objects

A timedelta object represents a duration, the difference between two dates or times.

```python
class datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
```

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

Only days, seconds and microseconds are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- 0 <= microseconds < 1000000
- 0 <= seconds < 3600*24 (the number of seconds in one day)
- -999999999 <= days <= 999999999

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range,
**OverflowError** is raised.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

**timedelta.min**
- The most negative *timedelta* object, `timedelta(-999999999)`.

**timedelta.max**
- The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

**timedelta.resolution**
- The smallest possible difference between non-equal *timedelta* objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max > -timedelta.min`. `-timedelta.max` is not representable as a *timedelta* object.

Instance attributes (read-only):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Between -999999999 and 999999999</td>
</tr>
</tbody>
</table>
### Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t1 = t2 + t3</code></td>
<td>Sum of <code>t2</code> and <code>t3</code>. Afterwards <code>t1-t2 == t3</code> and <code>t1-t3 == t2</code> are true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 - t3</code></td>
<td>Difference of <code>t2</code> and <code>t3</code>. Afterwards <code>t1 == t2 - t3</code> and <code>t2 == t1 + t3</code> are true. (1) (6)</td>
</tr>
<tr>
<td><code>t1 = t2 * i</code> or <code>t1 = i * t2</code></td>
<td>Delta multiplied by an integer. Afterwards <code>t1 // i == t2</code> is true, provided <code>i != 0</code>.</td>
</tr>
<tr>
<td></td>
<td>In general, <code>t1 * i == t1 * (i-1) + t1</code> is true. (1)</td>
</tr>
<tr>
<td><code>t1 = t2 * f</code> or <code>t1 = f * t2</code></td>
<td>Delta multiplied by a float. The result is rounded to the nearest multiple of <code>timedelta.resolution</code> using round-half-to-even.</td>
</tr>
<tr>
<td><code>f = t2 / t3</code></td>
<td>Division (3) of <code>t2</code> by <code>t3</code>. Returns a <code>float</code> object.</td>
</tr>
<tr>
<td><code>t1 = t2 / f</code> or <code>t1 = t2 / i</code></td>
<td>Delta divided by a float or an int. The result is rounded to the nearest multiple of <code>timedelta.resolution</code> using round-half-to-even.</td>
</tr>
<tr>
<td><code>t1 = t2 // i</code> or <code>t1 = t2 // t3</code></td>
<td>The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is</td>
</tr>
</tbody>
</table>

- `days` inclusive
- `seconds` Between 0 and 86399 inclusive
- `microseconds` Between 0 and 999999 inclusive
 returned. (3)

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = t_2 \ % t_3 )</td>
<td>The remainder is computed as a <code>timedelta</code> object. (3)</td>
</tr>
<tr>
<td><code>q, r = divmod(t1, t2)</code></td>
<td>Computes the quotient and the remainder: ( q = t_1 \ // \ t_2 ) (3) and ( r = t_1 \ % t_2 ). ( q ) is an integer and ( r ) is a <code>timedelta</code> object.</td>
</tr>
<tr>
<td><code>+t1</code></td>
<td>Returns a <code>timedelta</code> object with the same value. (2)</td>
</tr>
<tr>
<td><code>-t1</code></td>
<td>equivalent to <code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code>, and to ( t1* -1 ). (1)(4)</td>
</tr>
<tr>
<td><code>abs(t)</code></td>
<td>equivalent to <code>+t</code> when ( t).days ( &gt;= 0 ), and to <code>-t</code> when ( t).days &lt; 0. (2)</td>
</tr>
<tr>
<td><code>str(t)</code></td>
<td>Returns a string in the form ([D day[s], ][H]H:MM:SS[.UUUUUU]), where ( D ) is negative for negative ( t ). (5)</td>
</tr>
<tr>
<td><code>repr(t)</code></td>
<td>Returns a string representation of the <code>timedelta</code> object as a constructor call with canonical attribute values.</td>
</tr>
</tbody>
</table>

Notes:

1. This is exact, but may overflow.
2. This is exact, and cannot overflow.
3. Division by 0 raises `ZeroDivisionError`.
4. `-timedelta.max` is not representable as a `timedelta` object.
5. String representations of `timedelta` objects are normalized.
similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```python
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

6. The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when `t3` is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

In addition to the operations listed above `timedelta` objects support certain additions and subtractions with `date` and `datetime` objects (see below).

*Changed in version 3.2:* Floor division and true division of a `timedelta` object by another `timedelta` object are now supported, as are remainder operations and the `divmod()` function. True division and multiplication of a `timedelta` object by a `float` object are now supported.

Comparisons of `timedelta` objects are supported with the `timedelta` object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a `timedelta` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`timedelta` objects are hashable (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a `timedelta` object is
considered to be true if and only if it isn’t equal to `timedelta(0)`.

Instance methods:

`timedelta.total_seconds()`

Return the total number of seconds contained in the duration. Equivalent to `td / timedelta(seconds=1)`.

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

*New in version 3.2.*

Example usage:

```python
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=... minutes=50, seconds=600)
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(days=3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(days=3285), 9)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years
True
```
**date Objects**

A `date` object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

```
class datetime.date(year, month, day)
```

All arguments are required. Arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, `ValueError` is raised.

Other constructors, all class methods:

```
classmethod date.today()
```

Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

```
classmethod date.fromtimestamp(timestamp)
```

Return the local date corresponding to the POSIX timestamp, such as is returned by `time.time()`. This may raise
OverflowError, if the timestamp is out of the range of values supported by the platform C localtime() function, and OSError on localtime() failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp().

Changed in version 3.3: Raise OverflowError instead of ValueError if the timestamp is out of the range of values supported by the platform C localtime() function. Raise OSError instead of ValueError on localtime() failure.

classmethod date.fromordinal(ordinal)
Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. ValueError is raised unless 1 <= ordinal <= date.max.toordinal(). For any date d, date.fromordinal(d.toordinal()) == d.

classmethod date.fromisoformat(date_string)
Return a date corresponding to a date_string in the format emitted by date.isoformat(). Specifically, this function supports strings in the format(s) YYYY-MM-DD.

Caution: This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of date.isoformat().

New in version 3.7.

Class attributes:

date.min
The earliest representable date, \texttt{date(MINYEAR, 1, 1)}.

\texttt{date.max}

The latest representable date, \texttt{date(MAXYEAR, 12, 31)}.

\texttt{date.resolution}

The smallest possible difference between non-equal date objects, \texttt{timedelta(days=1)}.

Instance attributes (read-only):

\texttt{date.year}

Between \texttt{MINYEAR} and \texttt{MAXYEAR} inclusive.

\texttt{date.month}

Between 1 and 12 inclusive.

\texttt{date.day}

Between 1 and the number of days in the given month of the given year.

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{date2 = date1 + timedelta}</td>
<td>\texttt{date2} is \texttt{timedelta.days} days removed from \texttt{date1}. (1)</td>
</tr>
<tr>
<td>\texttt{date2 = date1 - timedelta}</td>
<td>Computes \texttt{date2} such that \texttt{date2 + timedelta == date1}. (2)</td>
</tr>
<tr>
<td>\texttt{timedelta = date1 - date2}</td>
<td>(3)</td>
</tr>
<tr>
<td>\texttt{date1 &lt; date2}</td>
<td>\texttt{date1} is considered less than \texttt{date2} when \texttt{date1} precedes \texttt{date2} in time. (4)</td>
</tr>
</tbody>
</table>
Notes:

1. \textit{date2} is moved forward in time if \texttt{timedelta.days > 0}, or backward if \texttt{timedelta.days < 0}. Afterward \texttt{date2 - date1} == \texttt{timedelta.days}. \texttt{timedelta.seconds} and \texttt{timedelta.microseconds} are ignored. \texttt{OverflowError} is raised if \texttt{date2.year} would be smaller than \texttt{MINYEAR} or larger than \texttt{MAXYEAR}.

2. \texttt{timedelta.seconds} and \texttt{timedelta.microseconds} are ignored.

3. This is exact, and cannot overflow. \texttt{timedelta.seconds} and \texttt{timedelta.microseconds} are 0, and \texttt{date2 + timedelta == date1} after.

4. In other words, \texttt{date1 < date2} if and only if \texttt{date1.toordinal() < date2.toordinal()}. Date comparison raises \texttt{TypeError} if the other comparand isn’t also a \texttt{date} object. However, \texttt{NotImplemented} is returned instead if the other comparand has a \texttt{timetuple()} attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a \texttt{date} object is compared to an object of a different type, \texttt{TypeError} is raised unless the comparison is \texttt{==} or \texttt{!=}. The latter cases return \texttt{False} or \texttt{True}, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all \texttt{date} objects are considered to be true.

Instance methods:

\texttt{date.replace(year=self.year, month=self.month, day=self.day)}

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if \texttt{d == date(2002, 12, 31)}, then
d.replace(day=26) == date(2002, 12, 26).

date().timetuple()
Return a `time.struct_time` such as returned by `time.localtime()`. The hours, minutes and seconds are 0, and the DST flag is -1. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st.

date().toordinal()
Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any `date` object `d`, `date.fromordinal(d.toordinal()) == d`.

date().weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `date(2002, 12, 4).weekday() == 2`, a Wednesday. See also `isoweekday()`.

date().isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also `weekday()`, `isocalendar()`.

date().isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian...
calendar. See [https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm](https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm) for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

date. `isof orm at()`
Return a string representing the date in ISO 8601 format, ‘YYYY-MM-DD’. For example, `date(2002, 12, 4).isof orm at() == '2002-12-04'`.

date. `__str__()`
For a date `d`, `str(d)` is equivalent to `d.isof orm at()`.

date. `ctime()`
Return a string representing the date, for example `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `date.ctime()` does not invoke) conforms to the C standard.
date. `strftime(format)`

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

date. `__format__(format)`

Same as `date.strftime()`. This makes it possible to specify a format string for a `date` object in formatted string literals and when using `str.format()`. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

Example of counting days to an event:

```python
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today = date.today().date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Example of working with `date`:

```python
>>> from datetime import date
>>> d = date.fromordinal(730920)  # 730920th day after 1.1.0001
```
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002   # year
3      # month
11     # day
0      # day of week (0 = Monday)
0      # day of year
0      # week number in year
70     # 70th day of year
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002   # ISO year
11     # ISO week number
1      # ISO day number (1 = Monday)
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}'.format(d, t[0], t[1])
'The day is 11, the month is March.'
**datetime** Objects

A **datetime** object is a single object containing all the information from a **date** object and a **time** object. Like a **date** object, **datetime** assumes the current Gregorian calendar extended in both directions; like a time object, **datetime** assumes there are exactly $3600 \times 24$ seconds in every day.

Constructor:

```python
class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)
```

The year, month and day arguments are required. `tzinfo` may be `None`, or an instance of a **tzinfo** subclass. The remaining arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= number of days in the given month and year`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, **ValueError** is raised.

*New in version 3.6:* Added the `fold` argument.

Other constructors, all class methods:
classmethod datetime.today()

Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also now(), fromtimestamp().

classmethod datetime.now(tz=None)

Return the current local date and time. If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more precision than can be gotten from going through a time.time() timestamp (for example, this may be possible on platforms supplying the C gettimeofday() function).

If tz is not None, it must be an instance of a tzinfo subclass, and the current date and time are converted to tz’s time zone. In this case the result is equivalent to tz.fromutc(datetime.utcnow().replace(tzinfo=tz)). See also today(), utcnow().

classmethod datetime.utcnow()

Return the current UTC date and time, with tzinfo None. This is like now(), but returns the current UTC date and time, as a naive datetime object. An aware current UTC datetime can be obtained by calling datetime.now(timezone.utc). See also now().

classmethod datetime.fromtimestamp(timestamp, tz=None)

Return the local date and time corresponding to the POSIX timestamp, such as is returned by time.time(). If optional argument tz is None or not specified, the timestamp is converted to the platform’s local date and time, and the returned datetime object is naive.
If tz is not None, it must be an instance of a tzinfo subclass, and the timestamp is converted to tz’s time zone. In this case the result is equivalent to

tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))

fromtimestamp() may raise OverflowError, if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions, and OSError on localtime() or gmtime() failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp(), and then it's possible to have two timestamps differing by a second that yield identical datetime objects. See also utcfromtimestamp().

Changed in version 3.3: Raise OverflowError instead of ValueError if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions. Raise OSError instead of ValueError on localtime() or gmtime() failure.

Changed in version 3.6: fromtimestamp() may return instances with fold set to 1.

classmethod datetime.utcfromtimestamp(timestamp)

Return the UTC datetime corresponding to the POSIX timestamp, with tzinfo None. This may raise OverflowError, if the timestamp is out of the range of values supported by the platform C gmtime() function, and OSError on gmtime() failure. It's common for this to be restricted to years in 1970 through 2038.
To get an aware `datetime` object, call `fromtimestamp()`:

```python
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```python
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta...
```

except the latter formula always supports the full years range: between `MINYEAR` and `MAXYEAR` inclusive.

*Changed in version 3.3:* Raise `OverflowError` instead of `ValueError` if the timestamp is out of the range of values supported by the platform C `gmtime()` function. Raise `OSError` instead of `ValueError` on `gmtime()` failure.

**classmethod `datetime.fromordinal(ordinal)`**

Return the `datetime` corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and `tzinfo` is `None`.

**classmethod `datetime.combine(date, time, tzinfo=self.tzinfo)`**

Return a new `datetime` object whose date components are equal to the given `date` object’s, and whose time components are equal to the given `time` object’s. If the `tzinfo` argument is provided, its value is used to set the `tzinfo` attribute of the result, otherwise the `tzinfo` attribute of the `time` argument is used.
For any `datetime` object `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`. If `date` is a `datetime` object, its time components and `tzinfo` attributes are ignored.

*Changed in version 3.6:* Added the `tzinfo` argument.

**classmethod datetime. `fromisoformat` (date_string)**

Return a `datetime` corresponding to a `date_string` in one of the formats emitted by `date.isoformat()` and `datetime.isoformat()`. Specifically, this function supports strings in the format(s) `YYYY-MM-DD[+HH[:MM[:SS[.fff[fff]]]]]` `YYY-MM-DDTHH[:MM[:SS[.ffffff]]]`, where `*` can match any single character.

**Caution:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of `datetime.isoformat()`.

*New in version 3.7.*

**classmethod datetime. `strptime` (date_string, format)**

Return a `datetime` corresponding to `date_string`, parsed according to `format`. This is equivalent to `datetime(*time.strptime(date_string, format)[0:6])`. `ValueError` is raised if the `date_string` and `format` can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

Class attributes:
datetime. **min**

The earliest representable datetime, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime. **max**

The latest representable datetime, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime. **resolution**

The smallest possible difference between non-equal datetime objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

datetime. **year**

Between MINYEAR and MAXYEAR inclusive.

datetime. **month**

Between 1 and 12 inclusive.

datetime. **day**

Between 1 and the number of days in the given month of the given year.

datetime. **hour**

In range(24).

datetime. **minute**

In range(60).

datetime. **second**

In range(60).
**datetime.microsecond**

In `range(1000000)`.

**datetime.tzinfo**

The object passed as the `tzinfo` argument to the `datetime` constructor, or `None` if none was passed.

**datetime.fold**

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

*New in version 3.6.*

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datetime2 = datetime1 + timedelta</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>datetime2 = datetime1 - timedelta</code></td>
<td>(2)</td>
</tr>
<tr>
<td><code>timedelta = datetime1 - datetime2</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>datetime1 &lt; datetime2</code></td>
<td>Compares <code>datetime</code> to <code>datetime</code>. (4)</td>
</tr>
</tbody>
</table>

1. `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same `tzinfo`
attribute as the input datetime, and datetime2 - datetime1 ==
timedelta after. OverflowError is raised if datetime2.year would
be smaller than MINYEAR or larger than MAXYEAR. Note that no
time zone adjustments are done even if the input is an aware
object.

2. Computes the datetime2 such that datetime2 + timedelta ==
datetime1. As for addition, the result has the same tzinfo
attribute as the input datetime, and no time zone adjustments are
done even if the input is aware.

3. Subtraction of a datetime from a datetime is defined only if
both operands are naive, or if both are aware. If one is aware and
the other is naive, TypeError is raised.

If both are naive, or both are aware and have the same tzinfo
attribute, the tzinfo attributes are ignored, and the result is a
timedelta object t such that datetime2 + t ==
datetime1. No time zone adjustments are done in this case.

If both are aware and have different tzinfo attributes, a-b acts
as if a and b were first converted to naive UTC datetimes first.
The result is (a.replace(tzinfo=None) -
a.utcoffset()) - (b.replace(tzinfo=None) -
b.utcoffset()) except that the implementation never
overflows.

4. datetime1 is considered less than datetime2 when datetime1
precedes datetime2 in time.

If one comparand is naive and the other is aware, TypeError is
raised if an order comparison is attempted. For equality
comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base datetimes are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

*Changed in version 3.3:* Equality comparisons between naive and aware `datetime` instances don’t raise `TypeError`.

**Note:** In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises `TypeError` if the other comparand isn’t also a `datetime` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `datetime` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`datetime` objects can be used as dictionary keys. In Boolean contexts, all `datetime` objects are considered to be true.

Instance methods:

`datetime.date()`

Return `date` object with same year, month and day.

`datetime.time()`
Return `time` object with same hour, minute, second, microsecond and fold. `tzinfo` is `None`. See also method `timetz()`.

*Changed in version 3.6:* The fold value is copied to the returned `time` object.

datetime.`timetz()`

Return `time` object with same hour, minute, second, microsecond, fold, and `tzinfo` attributes. See also method `time()`.

*Changed in version 3.6:* The fold value is copied to the returned `time` object.

datetime.`replace`(year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *fold=0)

Return a `datetime` with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive `datetime` from an aware `datetime` with no conversion of date and time data.

*New in version 3.6:* Added the `fold` argument.

datetime.`astimezone`(tz=None)

Return a `datetime` object with new `tzinfo` attribute `tz`, adjusting the date and time data so the result is the same UTC time as `self`, but in `tz`'s local time.

If provided, `tz` must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. If `self` is naive, it is presumed to represent time in the system timezone.
If called without arguments (or with `tz=None`) the system local timezone is assumed for the target timezone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of `timezone` with the zone name and offset obtained from the OS.

If `self.tzinfo` is `tz`, `self.astimezone(tz)` is equal to `self`: no adjustment of date or time data is performed. Else the result is local time in the timezone tz, representing the same UTC time as `self`: after `astz = dt.astimezone(tz), astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a time zone object `tz` to a datetime `dt` without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the time zone object from an aware datetime `dt` without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Note that the default `tzinfo.fromutc()` method can be overridden in a `tzinfo` subclass to affect the result returned by `astimezone()`. Ignoring error cases, `astimezone()` acts like:

```python
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone.
    utc = (self - self.utcoffset()).replace(tzinfo=self.tzinfo)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

*Changed in version 3.3:* `tz` now can be omitted.
Changed in version 3.6: The `astimezone()` method can now be called on naive instances that are presumed to represent system local time.

```python
datetime.utcoffset()
```

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn't return `None` or a `timedelta` object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

```python
datetime.dst()
```

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn't return `None` or a `timedelta` object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

```python
datetime.tzname()
```

If `tzinfo` is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn't return `None` or a string object,

```python
datetime.timetuple()
```

Return a `time.struct_time` such as returned by `time.localtime(). d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst)), where`
yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The tm_isdst flag of the result is set according to the dst() method: tzinfo is None or dst() returns None, tm_isdst is set to -1; else if dst() returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.utctimetuple()
If datetime instance d is naive, this is the same as d.timetuple() except that tm_isdst is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If d is aware, d is normalized to UTC time, by subtracting d.utcoffset(), and a time.struct_time for the normalized time is returned. tm_isdst is forced to 0. Note that an OverflowError may be raised if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

datetime.toordinal()
Return the proleptic Gregorian ordinal of the date. The same as self.date().toordinal().

datetime.timestamp()
Return POSIX timestamp corresponding to the datetime instance. The return value is a float similar to that returned by time.time().

Naive datetime instances are assumed to represent local time and this method relies on the platform C mktime() function to perform the conversion. Since datetime supports wider range of values than mktime() on many platforms, this method may raise
OverflowError for times far in the past or far in the future.

For aware datetime instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

New in version 3.3.

Changed in version 3.6: The timestamp() method uses the fold attribute to disambiguate the times during a repeated interval.

**Note:** There is no method to obtain the POSIX timestamp directly from a naive datetime instance representing UTC time. If your application uses this convention and your system timezone is not set to UTC, you can obtain the POSIX timestamp by supplying tzinfo=timezone.utc:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```
or by calculating the timestamp directly:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(datetime.weekday())
```

datetime.weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as self.date().weekday(). See also isoweekday().

datetime.isoweekday()

Return the day of the week as an integer, where Monday is 1 and
Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday(), isocalendar()`.

**datetime. isocalendar()**

Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

**datetime. isoformat(sep='T', timespec='auto')**

Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.ffffff or, if `microsecond` is 0, YYYY-MM-DDTHH:MM:SS

If `utcoffset()` does not return `None`, a string is appended, giving the UTC offset: YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[SS].fff or, if `microsecond` is 0 YYYY-MM-DDTHH:MM:SS+HH:MM[SS].fff.

The optional argument `sep` (default 'T') is a one-character separator, placed between the date and time portions of the result. For example,

```python
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat
'2002-12-25 00:00:00-06:39'
```

The optional argument `timespec` specifies the number of additional components of the time to include (the default is 'auto'). It can be one of the following:

- 'auto': Same as 'seconds' if `microsecond` is 0, same
as 'microseconds' otherwise.

- 'hours': Include the hour in the two-digit HH format.
- 'minutes': Include hour and minute in HH:MM format.
- 'seconds': Include hour, minute, and second in HH:MM:SS format.
- 'milliseconds': Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- 'microseconds': Include full time in HH:MM:SS.fffffff format.

**Note:** Excluded time components are truncated, not rounded.

*ValueError* will be raised on an invalid *timespec* argument.

```python
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

*New in version 3.6:* Added the *timespec* argument.

`datetime.__str__()`

For a `datetime` instance `d`, `str(d)` is equivalent to `d.isoformat(' ')`.

`datetime.ctime()`

Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `datetime.ctime()` does not invoke)
conforms to the C standard.

datetime. **strftime**(format)

Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see **strftime()** and **strptime()** Behavior.

datetime. **__format__**(format)

Same as `datetime.strftime()`. This makes it possible to specify a format string for a `datetime` object in formatted string literals and when using `str.format()`. For a complete list of formatting directives, see **strftime()** and **strptime()** Behavior.

Examples of working with datetime objects:

```python
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)
>>> datetime.datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...```
Using datetime with tzinfo:

```python
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)  # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday())
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday())
...         if self.dston <= dt.replace(tzinfo=None):
```

```python
...             return self.dstoff
...         else:
...             return self.dston
...     dst = dst
...     # tzinfo calculations
...     tzname = str
...    tzinfo = None
```
...            return timedelta(hours=1)
...        else:
...            return timedelta(0)
...
    def tzname(self, dt):
        return "GMT +1"

>>> class GMT2(tzinfo):
...    def utcoffset(self, dt):
...        return timedelta(hours=2) + self.dst(dt)
...    def dst(self, dt):
...        d = datetime(dt.year, 4, 1)
...        self.dston = d - timedelta(days=d.weekday)
...        d = datetime(dt.year, 11, 1)
...        self.dstoff = d - timedelta(days=d.weekday)
...        if self.dston <= dt.replace(tzinfo=None):
...            return timedelta(hours=1)
...        else:
...            return timedelta(0)
...
    def tzname(self, dt):
        return "GMT +2"

>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(seconds=3600)

>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(seconds=3600)
>>> dt2.utcoffset()
datetime.timedelta(seconds=7200)

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())

>>> dt3  # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>
>>> dt2  # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
**time** Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a `tzinfo` object.

```python
class datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0)
```

All arguments are optional. `tzinfo` may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments may be integers, in the following ranges:

- $0 \leq hour < 24$,
- $0 \leq minute < 60$,
- $0 \leq second < 60$,
- $0 \leq microsecond \leq 1000000$,
- `fold` in $[0, 1]$.

If an argument outside those ranges is given, `ValueError` is raised. All default to $0$ except `tzinfo`, which defaults to `None`.

Class attributes:

- **time.min**
  
  The earliest representable `time`, `time(0, 0, 0, 0)`.

- **time.max**
  
  The latest representable `time`, `time(23, 59, 59, 999999)`.

- **time.resolution**
  
  The smallest possible difference between non-equal `time` objects, `timedelta(microseconds=1)`, although note that arithmetic on `time` objects is not supported.
Instance attributes (read-only):

```
    time.hour
        In range(24).

    time.minute
        In range(60).

    time.second
        In range(60).

    time.microsecond
        In range(1000000).

    time.tzinfo
        The object passed as the tzinfo argument to the time constructor, or None if none was passed.

    time.fold
        In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

        New in version 3.6.
```

Supported operations:

- comparison of `time` to `time`, where `a` is considered less than `b` when `a` precedes `b` in time. If one comparand is naive and the other is aware, `TypeError` is raised if an order comparison is
attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same tzinfo attribute, the common tzinfo attribute is ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a time object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively.

*Changed in version 3.3:* Equality comparisons between naive and aware time instances don’t raise TypeError.

- hash, use as dict key
- efficient pickling

In boolean contexts, a time object is always considered to be true.

*Changed in version 3.5:* Before Python 3.5, a time object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

Other constructor:

**classmethod time. fromisoformat(time_string)**

Return a time corresponding to a time_string in one of the formats emitted by time.isoformat(). Specifically, this function
supports strings in the format(s) HH[:MM[:SS[.fff[fff]]]]
[+HH:MM[:SS[.ffffff]]].

**Caution:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of `time.isoformat()`.

*New in version 3.7.*

Instance methods:

```python
time.replace(hour=self.hour, minute=self.minute,
second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo,
* fold=0)
```

Return a `time` with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive `time` from an aware `time`, without conversion of the time data.

*New in version 3.6:* Added the `fold` argument.

```python
time.isoformat(timespec='auto')
```

Return a string representing the time in ISO 8601 format, HH:MM:SS.ffffff or, if `microsecond` is 0, HH:MM:SS If `utcoffset()` does not return None, a string is appended, giving the UTC offset: HH:MM:SS.ffffff+HH:MM:[SS[.ffffff]] or, if self.microsecond is 0, HH:MM:SS+HH:MM:[SS[.ffffff]].

The optional argument `timespec` specifies the number of additional components of the time to include (the default is 'auto'). It can be one of the following:

- 'auto': Same as 'seconds' if `microsecond` is 0, same
as 'microseconds' otherwise.
- 'hours': Include the **hour** in the two-digit HH format.
- 'minutes': Include **hour** and **minute** in HH:MM format.
- 'seconds': Include **hour**, **minute**, and **second** in HH:MM:SS format.
- 'milliseconds': Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- 'microseconds': Include full time in HH:MM:SS.ffffff format.

**Note:** Excluded time components are truncated, not rounded.

**ValueError** will be raised on an invalid *timespec* argument.

```python
>>> from datetime import time
generate the following in the next block:
```

```python
>>> time(hour=12, minute=34, second=56, microsecond='12:34'
```

```python
>>> dt = time(hour=12, minute=34, second=56, microsecond='12:34:56.000000'
```

```python
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

*New in version 3.6:* Added the *timespec* argument.

**time.**`str()`

For a time `t`, `str(t)` is equivalent to `t.isoformat()`.

**time.**`strftime(format)`

Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

**time.**`format__(format)`

Same as `time.strftime()`. This makes it possible to specify a
format string for a time object in formatted string literals and when using str.format(). For a complete list of formatting directives, see strftime() and strptime() Behavior.

time.utcoffset()
If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None), and raises an exception if the latter doesn’t return None or a timedelta object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

time.dst()
If tzinfo is None, returns None, else returns self.tzinfo.dst(None), and raises an exception if the latter doesn’t return None, or a timedelta object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

time.tzname()
If tzinfo is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception if the latter doesn’t return None or a string object.

Example:

```python
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
```
```python
... def dst(self, dt):
...     return timedelta(0)
...
... def tzname(self, dt):
...     return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}'.format("time", t)
'The time is 12:10.'
```
tzinfo Objects

class datetime.tzinfo

This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard tzinfo methods needed by the datetime methods you use. The datetime module supplies a simple concrete subclass of tzinfo, timezone, which can represent timezones with fixed offset from UTC such as UTC itself or North American EST and EDT.

An instance of (a concrete subclass of) tzinfo can be passed to the constructors for datetime and time objects. The latter objects view their attributes as being in local time, and the tzinfo object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A tzinfo subclass must have an __init__() method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of tzinfo may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware datetime objects. If in doubt, simply implement all of them.

tzinfo.utc_offset(dt)

Return offset of local time from UTC, as a timedelta object that
is positive east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a tzinfo object represents both time zone and DST adjustments, utcoffset() should return their sum. If the UTC offset isn't known, return None. Else the value returned must be a timedelta object strictly between -timedelta(hours=24) and timedelta(hours=24) (the magnitude of the offset must be less than one day). Most implementations of utcoffset() will probably look like one of these two:

```
return CONSTANT  # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware
```

If utcoffset() does not return None, dst() should not return None either.

The default implementation of utcoffset() raises NotImplementedError.

*Changed in version 3.7:* The UTC offset is not restricted to a whole number of minutes.

tzinfo.dst(dt)

Return the daylight saving time (DST) adjustment, as a timedelta object or None if DST information isn’t known. Return timedelta(0) if DST is not in effect. If DST is in effect, return the offset as a timedelta object (see utcoffset() for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by utcoffset(), so there’s no need to consult dst() unless you’re interested in obtaining DST info.
separately. For example, `datetime.timetuple()` calls its `tzinfo` attribute's `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance `tz` of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime dt` with `dt.tzinfo == tz` For sane `tzinfo` subclasses, this expression yields the time zone’s “standard offset”, which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:

```python
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```

or

```python
def dst(self, dt):
    # Code to set dston and dstoff to the time zone
    # transition times based on the input dt.year
```
The default implementation of `dst()` raises `NotImplementedError`.

*Changed in version 3.7:* The DST offset is not restricted to a whole number of minutes.

tzinfo.tzname(dt)

Return the time zone name corresponding to the datetime object `dt`, as a string. Nothing about string names is defined by the `datetime` module, and there's no requirement that it mean anything in particular. For example, “GMT”, “UTC”, “-500”, “-5:00”, “EDT”, “US/Eastern”, “America/New York” are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some tzinfo subclasses will wish to return different names depending on the specific value of `dt` passed, especially if the tzinfo class is accounting for daylight time.

The default implementation of `tzname()` raises `NotImplementedError`.

These methods are called by a datetime or time object, in response to their methods of the same names. A datetime object passes itself as the argument, and a time object passes `None` as the argument. A tzinfo subclass’s methods should therefore be
prepared to accept a dt argument of None, or of class datetime.

When None is passed, it’s up to the class designer to decide the best response. For example, returning None is appropriate if the class wishes to say that time objects don’t participate in the tzinfo protocols. It may be more useful for utcoffset(None) to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a datetime object is passed in response to a datetime method, dt.tzinfo is the same object as self.tzinfo methods can rely on this, unless user code calls tzinfo methods directly. The intent is that the tzinfo methods interpret dt as being in local time, and not need worry about objects in other timezones.

There is one more tzinfo method that a subclass may wish to override:

tzinfo.fromutc(dt)

This is called from the default datetime.astimezone() implementation. When called from that, dt.tzinfo is self, and dt’s date and time data are to be viewed as expressing a UTC time. The purpose of fromutc() is to adjust the date and time data, returning an equivalent datetime in self’s local time.

Most tzinfo subclasses should be able to inherit the default fromutc() implementation without problems. It’s strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default fromutc() implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on
the specific date and time passed, which can happen for political reasons. The default implementations of \texttt{astimezone()} and \texttt{fromutc()} may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default \texttt{fromutc()} implementation acts like:

```python
def fromutc(self, dt):
    # raise ValueError error if \texttt{dt.tzinfo} is not \texttt{self}.
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if \texttt{dtoff} is None or \texttt{dtdst}.
    delta = dtoff - dtdst  # this is \texttt{self}'s standard offset
    if delta:
        dt += delta  # convert to standard local
        dtdst = dt.dst()
        # raise ValueError if \texttt{dtdst} is None.
        if dtdst:
            return dt + dtdst
        else:
            return dt
```

In the following \texttt{tzinfo_examples.py} file there are some examples of \texttt{tzinfo} classes:

```python
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
```
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):
    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)).total_seconds()
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond, tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO
```python
def tzname(self, dt):
    return _time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (dt.year, dt.month, dt.day,
          dt.hour, dt.minute, dt.second,
          dt.weekday(), 0, 0)
    stamp = _time.mktime(tt)
    tt = _time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases)
# DST start and end times. For a complete and up-to-
# and timezone definitions, visit the Olson Database
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be
# In the US, since 2007, DST starts at 2am (standard
# Sunday in March, which is the first Sunday on or a
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (stand
# Sunday in April and to end at 2am (DST time) on the
# Sunday of October, which is the first Sunday on or
# after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)

# From 1967 to 1986, DST used to start at 2am (standard time)
# on the last Sunday in April (the one on or after April 24) and
# on the last Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967,
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
    elif 1966 < year < 1987:
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end

class USTimeZone(tzinfo):
    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
return self.reprname

def tzname(self, dt):
    if self.dst(dt):
        return self.dstname
    else:
        return self.stdname

def utcoffset(self, dt):
    return self.stdoffset + self.dst(dt)

def dst(self, dt):
    if dt is None or dt.tzinfo is None:
        # An exception may be sensible here, in
        # It depends on how you want to treat the
        # fromutc() implementation (called by the
        # implementation) passes a datetime with
        return ZERO
    assert dt.tzinfo is self
    start, end = us_dst_range(dt.year)
    # Can't compare naive to aware objects, so strip
    # dt first.
    dt = dt.replace(tzinfo=None)
    if start + HOUR <= dt < end - HOUR:
        # DST is in effect.
        return HOUR
    if end - HOUR <= dt < end:
        # Fold (an ambiguous hour): use dt.fold
        return ZERO if dt.fold else HOUR
    if start <= dt < start + HOUR:
        # Gap (a non-existent hour): reverse the
        return HOUR if dt.fold else ZERO
    # DST is off.
    return ZERO

def fromutc(self, dt):
    assert dt.tzinfo is self
start, end = us_dst_range(dt.year)
start = start.replace(tzinfo=self)
dst_time = dt + self.stdoffset
std_time = std_time + HOUR
if end <= dst_time < end + HOUR:
    # Repeated hour
    return std_time.replace(fold=1)
if std_time < start or dst_time >= end:
    # Standard time
    return std_time
if start <= std_time < end - HOUR:
    # Daylight saving time
    return dst_time

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

Note that there are unavoidable subtleties twice per year in a tzinfo subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

<table>
<thead>
<tr>
<th>Time</th>
<th>UTC</th>
<th>EST</th>
<th>EDT</th>
<th>Pacific</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:MM</td>
<td>23:MM</td>
<td>0:MM</td>
<td>1:MM</td>
<td>2:MM</td>
</tr>
<tr>
<td>5:MM</td>
<td>0:MM</td>
<td>1:MM</td>
<td>2:MM</td>
<td>3:MM</td>
</tr>
<tr>
<td>8:MM</td>
<td>22:MM</td>
<td>23:MM</td>
<td>0:MM</td>
<td>1:MM</td>
</tr>
</tbody>
</table>

When DST starts (the “start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won’t deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get

```python
>>> from datetime import datetime
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
... 05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the “end” line), there’s a potentially worse problem: there’s an hour that can’t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that’s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock’s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the `fold` attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get

```python
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
... 05:00:00 UTC = 00:00:00 EDT
06:00:00 UTC = 01:00:00 EDT
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```
... u = u0 + i*HOUR
... t = u.astimezone(Eastern)
... print(u.time(), 'UTC =', t.time(), t.tzname()
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0

Note that the datetime instances that differ only by the value of the fold attribute are considered equal in comparisons.

Applications that can't bear wall-time ambiguities should explicitly check the value of the fold attribute or avoid using hybrid tzinfo subclasses; there are no ambiguities when using timezone, or any other fixed-offset tzinfo subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See also:

dateutil.tz
The standard library has timezone class for handling arbitrary fixed offsets from UTC and timezone.utc as UTC timezone instance.

dateutil.tz library brings the IANA timezone database (also known as the Olson database) to Python and its usage is recommended.

IANA timezone database
The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.
timezone Objects

The `timezone` class is a subclass of `tzinfo`, each instance of which represents a timezone defined by a fixed offset from UTC. Note that objects of this class cannot be used to represent timezone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

```python
class datetime.timezone(offset, name=None)
```

- The `offset` argument must be specified as a `timedelta` object representing the difference between the local time and UTC. It must be strictly between \(-\text{timedelta(hours}=24)\) and \(\text{timedelta(hours}=24)\), otherwise `ValueError` is raised.

- The `name` argument is optional. If specified it must be a string that will be used as the value returned by the `datetime.tzname()` method.

  *New in version 3.2.*

  *Changed in version 3.7:* The UTC offset is not restricted to a whole number of minutes.

```python
timezone.utcoffset(dt)
```

- Return the fixed value specified when the `timezone` instance is constructed. The `dt` argument is ignored. The return value is a `timedelta` instance equal to the difference between the local time and UTC.

  *Changed in version 3.7:* The UTC offset is not restricted to a whole number of minutes.
timezone.tzname(dt)
Return the fixed value specified when the timezone instance is constructed. If name is not provided in the constructor, the name returned by tzname(dt) is generated from the value of the offset as follows. If offset is timedelta(0), the name is “UTC”, otherwise it is a string ‘UTC±HH:MM’, where ± is the sign of offset, HH and MM are two digits of offset.hours and offset.minutes respectively.

*Changed in version 3.6:* Name generated from offset=timedelta(0) is now plain ‘UTC’, not ‘UTC+00:00’.

timezone.dst(dt)
Always returns None.

timezone.fromutc(dt)
Return dt + offset. The dt argument must be an aware datetime instance, with tzinfo set to self.

Class attributes:

timezone.utc
The UTC timezone, timezone(timedelta(0)).
strftime() and strptime() Behavior

date, datetime, and time objects all support a strftime(format) method, to create a string representing the time under the control of an explicit format string. Broadly speaking, d.strftime(fmt) acts like the time module's time.strftime(fmt, d.timetuple()) although not all objects support a timetuple() method.

Conversely, the datetime.strptime() class method creates a datetime object from a string representing a date and time and a corresponding format string. datetime.strptime(date_string, format) is equivalent to datetime(* (time.strptime(date_string, format)[0:6])), except when the format includes sub-second components or timezone offset information, which are supported in datetime.strptime but are discarded by time.strptime.

For time objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For date objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as date objects have no such values. If they're used anyway, 0 is substituted for them.

The full set of format codes supported varies across platforms, because Python calls the platform C library's strftime() function, and platform variations are common. To see the full set of format codes supported on your platform, consult the strftime(3) documentation.
For the same reason, handling of format strings containing Unicode code points that can’t be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others \texttt{strfti}me may raise \texttt{UnicodeError} or return an empty string instead.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format codes.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday as locale’s abbreviated name.</td>
<td>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday as locale’s full name.</td>
<td>Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.</td>
<td>0, 1, ..., 6</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal number.</td>
<td>01, 02, ..., 31</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Month as locale's abbreviated name.</td>
<td>Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Month as locale's full name.</td>
<td>January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a zero-padded decimal number.</td>
<td>00, 01, ..., 99</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td>0001, 0002, ..., 2013, 2014, ..., 9998, 9999</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>00, 01, ..., 23</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale's equivalent of either AM or PM.</td>
<td>AM, PM (en_US); am, pm (de_DE)</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td>Values</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td></td>
</tr>
<tr>
<td>%f</td>
<td>Microsecond as a decimal number, zero-padded on the left.</td>
<td>000000, 000001, ..., 999999</td>
<td></td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive).</td>
<td>(empty), +0000, -0400, +1030, +063415, -030712.345216</td>
<td></td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name (empty string if the object is naive).</td>
<td>(empty), UTC, EST, CST</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a zero-padded decimal number.</td>
<td>001, 002, ..., 366</td>
<td></td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>00, 01, ..., 53</td>
<td></td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td>00, 01, ..., 53</td>
<td></td>
</tr>
<tr>
<td>Locale's appropriate date</td>
<td>Tue Aug 16 21:30:00 1988</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values. These may not be available on all platforms when used with the `strftime()` method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling `strptime()` with incomplete or ambiguous ISO 8601 directives will raise a `ValueError`.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>and time representation.</td>
<td>(en_US); Di 16 Aug 21:30:00 1988 (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td>08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
<td>21:30:00 (en_US); 21:30:00 (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%%</td>
<td>A literal '%' character.</td>
<td>%</td>
<td></td>
</tr>
</tbody>
</table>
is Monday.

| %V      | ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4. | 01, 02, ..., 53 | (8) |

New in version 3.6: %G, %u and %V were added.

Notes:

1. Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, “month/day/year” versus “day/month/year”), and the output may contain Unicode characters encoded using the locale’s default encoding (for example, if the current locale is ja_JP, the default encoding could be any one of eucJP, SJIS, or utf-8; use locale.getlocale() to determine the current locale’s encoding).

2. The strptime() method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

   Changed in version 3.2: In previous versions, strftime() method was restricted to years >= 1900.

   Changed in version 3.3: In version 3.2, strftime() method was restricted to years >= 1000.

3. When used with the strptime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.
4. Unlike the `time` module, the `datetime` module does not support leap seconds.

5. When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

6. For a naive object, the `%z` and `%Z` format codes are replaced by empty strings.

   For an aware object:

   `%z`

   `utcoffset()` is transformed into a string of the form ±HHMM[SS[.ffffff]], where HH is a 2-digit string giving the number of UTC offset hours, MM is a 2-digit string giving the number of UTC offset minutes, SS is a 2-digit string giving the number of UTC offset seconds and fffffff is a 6-digit string giving the number of UTC offset microseconds. The fffffff part is omitted when the offset is a whole number of seconds and both the fffffff and the SS part is omitted when the offset is a whole number of minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string '-0330'.

   `Changed in version 3.7:` The UTC offset is not restricted to a whole number of minutes.

   `Changed in version 3.7:` When the `%z` directive is provided to the `strptime()` method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example,
'+01:00:00' will be parsed as an offset of one hour. In addition, providing 'Z' is identical to '+00:00'.

%Z

If tzname() returns None, %Z is replaced by an empty string. Otherwise %Z is replaced by the returned value, which must be a string.

Changed in version 3.2: When the %z directive is provided to the strftime() method, an aware datetime object will be produced. The tzinfo of the result will be set to a timezone instance.

7. When used with the strftime() method, %U and %W are only used in calculations when the day of the week and the calendar year (%Y) are specified.

8. Similar to %U and %W, %V is only used in calculations when the day of the week and the ISO year (%G) are specified in a strftime() format string. Also note that %G and %Y are not interchangeable.

Footnotes

[1] If, that is, we ignore the effects of Relativity
calendar — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix cal program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use `setfirstweekday()` to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the datetime and time modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book “Calendrical Calculations”, where it’s the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

```python
class calendar.Calendar(firstweekday=0)

Creates a Calendar object. firstweekday is an integer specifying the first day of the week. 0 is Monday (the default), 6 is Sunday.

A Calendar object provides several methods that can be used for preparing the calendar data for formatting. This class doesn’t do any formatting itself. This is the job of subclasses.
```
Calendar instances have the following methods:

**iterweekdays()**
Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the firstweekday property.

**itermonthdates(year, month)**
Return an iterator for the month month (1–12) in the year year. This iterator will return all days (as datetime.date objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

**itermonthdays(year, month)**
Return an iterator for the month month in the year year similar to itermonthdates(), but not restricted by the datetime.date range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is 0.

**itermonthdays2(year, month)**
Return an iterator for the month month in the year year similar to itermonthdates(), but not restricted by the datetime.date range. Days returned will be tuples consisting of a day of the month number and a week day number.

**itermonthdays3(year, month)**
Return an iterator for the month month in the year year similar to itermonthdates(), but not restricted by the datetime.date range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.
New in version 3.7.

```python
itermonthdays4(year, month)
```
Return an iterator for the month `month` in the year `year` similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

New in version 3.7.

```python
monthdatescalendar(year, month)
```
Return a list of the weeks in the month `month` of the `year` as full weeks. Weeks are lists of seven `datetime.date` objects.

```python
monthdays2calendar(year, month)
```
Return a list of the weeks in the month `month` of the `year` as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

```python
monthdayscalendar(year, month)
```
Return a list of the weeks in the month `month` of the `year` as full weeks. Weeks are lists of seven day numbers.

```python
yeardatescalendar(year, width=3)
```
Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to `width` months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are `datetime.date` objects.

```python
yeardays2calendar(year, width=3)
```
Return the data for the specified year ready for formatting
(similar to `yeardatescalendar()`). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

**yeardayscalendar**(*year, width=3*)

Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are day numbers. Day numbers outside this month are zero.

**class calendar. TextCalendar**(*firstweekday=0*)

This class can be used to generate plain text calendars.

**TextCalendar** instances have the following methods:

- **formatmonth**(theyear, themonth, w=0, l=0)
  Return a month’s calendar in a multi-line string. If w is provided, it specifies the width of the date columns, which are centered. If l is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

- **prmonth**(theyear, themonth, w=0, l=0)
  Print a month’s calendar as returned by `formatmonth()`.

- **formatyear**(theyear, w=2, l=1, c=6, m=3)
  Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method. The earliest year for which a calendar can be generated is platform-dependent.
pryear(theyear, w=2, l=1, c=6, m=3)
Print the calendar for an entire year as returned by formatyear()

class calendar.HTMLCalendar(firstweekday=0)
This class can be used to generate HTML calendars.

HTMLCalendar instances have the following methods:

formatmonth(theyear, themonth, withyear=True)
Return a month’s calendar as an HTML table. If withyear is true
the year will be included in the header, otherwise just the month
name will be used.

formatyear(theyear, width=3)
Return a year’s calendar as an HTML table. width (defaulting to
3) specifies the number of months per row.

formatyearpage(theyear, width=3, css='calendar.css',
encoding=None)
Return a year’s calendar as a complete HTML page. width
(defaulting to 3) specifies the number of months per row. css is
the name for the cascading style sheet to be used. None can
be passed if no style sheet should be used. encoding specifies
the encoding to be used for the output (defaulting to the system
default encoding).

HTMLCalendar has the following attributes you can override to
customize the CSS classes used by the calendar:

cssclasses
A list of CSS classes used for each weekday. The default class
list is:
more styles can be added for each day:

```python
cssclasses = ["mon", "tue", "wed", "thu", "fri", "mon text-bold", "tue", "wed", "thu"]
```

Note that the length of this list must be seven items.

**cssclass_noday**
The CSS class for a weekday occurring in the previous or coming month.

*New in version 3.7.*

**cssclasses_weekday_head**
A list of CSS classes used for weekday names in the header row. The default is the same as **cssclasses**.

*New in version 3.7.*

**cssclass_month_head**
The month's head CSS class (used by `formatmonthname()`). The default value is "month".

*New in version 3.7.*

**cssclass_month**
The CSS class for the whole month's table (used by `formatmonth()`). The default value is "month".

*New in version 3.7.*

**cssclass_year**
The CSS class for the whole year's table of tables (used by \texttt{formatyear()}). The default value is "year".

\textit{New in version 3.7.}

cssclass\_year\_head

The CSS class for the table head for the whole year (used by \texttt{formatyear()}). The default value is "year".

\textit{New in version 3.7.}

Note that although the naming for the above described class attributes is singular (e.g. cssclass\_month cssclass\_noday), one can replace the single CSS class with a space separated list of CSS classes, for example:

\begin{verbatim}
"text-bold text-red"
\end{verbatim}

Here is an example how HTMLCalendar can be customized:

\begin{verbatim}
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style]
class_month_head = "text-center month-head"
class_month = "text-center month"
class_year = "text-italic lead"
\end{verbatim}

class calendar\_LocaleTextCalendar(firstweekday=0, locale=None)

This subclass of TextCalendar can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.
class calendar.**LocaleHTMLCalendar**(*firstweekday=0, locale=None*)

This subclass of **HTMLCalendar** can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

**Note:** The **formatweekday()** and **formatmonthname()** methods of these two classes temporarily change the current locale to the given **locale**. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

**calendar.**setfirstweekday**(weekday)**

Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For example, to set the first weekday to Sunday:

```python
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

**calendar.**firstweekday**()**

Returns the current setting for the weekday to start each week.

**calendar.**isleap**(year)**

Returns **True** if **year** is a leap year, otherwise **False**.

**calendar.**leapdays**(y1, y2)**

Returns the number of leap years in the range from **y1** to **y2** (exclusive), where **y1** and **y2** are years.
This function works for ranges spanning a century change.

```python
calendar.weekday(year, month, day)
```
Returns the day of the week (0 is Monday) for `year` (1970–...), `month` (1–12), `day` (1–31).

```python
calendar.weekheader(n)
```
Returns a header containing abbreviated weekday names. `n` specifies the width in characters for one weekday.

```python
calendar.monthrange(year, month)
```
Returns weekday of first day of the month and number of days in month, for the specified `year` and `month`.

```python
calendar.monthcalendar(year, month)
```
Returns a matrix representing a month’s calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by `setfirstweekday()`.

```python
calendar.prmouth(year, month, w=0, l=0)
```
Prints a month’s calendar as returned by `month()`.

```python
calendar.month(year, month, w=0, l=0)
```
Returns a month’s calendar in a multi-line string using the `formatmonth()` of the `TextCalendar` class.

```python
calendar.prcal(year, w=0, l=0, c=6, m=3)
```
Prints the calendar for an entire year as returned by `calendar()`.

```python
calendar.calendar(year, w=2, l=1, c=6, m=3)
```
Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.
calendar.\texttt{timegm}(tuple)

An unrelated but handy function that takes a time tuple such as returned by the \texttt{gmtime()} function in the \texttt{time} module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, \texttt{time.gmtime()} and \texttt{timegm()} are each others' inverse.

The \texttt{calendar} module exports the following data attributes:

\begin{description}
\item[calendar.\texttt{day_name}]
An array that represents the days of the week in the current locale.
\item[calendar.\texttt{day_abbr}]
An array that represents the abbreviated days of the week in the current locale.
\item[calendar.\texttt{month_name}]
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and \texttt{month_name[0]} is the empty string.
\item[calendar.\texttt{month_abbr}]
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and \texttt{month_abbr[0]} is the empty string.
\end{description}

\textit{See also:}

\textbf{Module} \texttt{datetime}

Object-oriented interface to dates and times with similar functionality to the \texttt{time} module.
Module `time`

Low-level time related functions.
collections — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>namedtuple()</code></td>
<td>factory function for creating tuple subclasses with named fields</td>
</tr>
<tr>
<td><code>deque</code></td>
<td>list-like container with fast appends and pops on either end</td>
</tr>
<tr>
<td><code>ChainMap</code></td>
<td>dict-like class for creating a single view of multiple mappings</td>
</tr>
<tr>
<td><code>Counter</code></td>
<td>dict subclass for counting hashable objects</td>
</tr>
<tr>
<td><code>OrderedDict</code></td>
<td>dict subclass that remembers the order entries were added</td>
</tr>
<tr>
<td><code>defaultdict</code></td>
<td>dict subclass that calls a factory function to supply missing values</td>
</tr>
<tr>
<td><code>UserDict</code></td>
<td>wrapper around dictionary objects for easier dict subclassing</td>
</tr>
<tr>
<td><code>UserList</code></td>
<td>wrapper around list objects for easier list subclassing</td>
</tr>
<tr>
<td><code>UserString</code></td>
<td>wrapper around string objects for easier string subclassing</td>
</tr>
</tbody>
</table>

*Changed in version 3.3:* Moved Collections Abstract Base Classes to
the `collections.abc` module. For backwards compatibility, they continue to be visible in this module through Python 3.7. Subsequently, they will be removed entirely.
ChainMap objects

New in version 3.3.

A ChainMap class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.ChainMap(*maps)

A ChainMap groups multiple dicts or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the maps attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A ChainMap incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.

All of the usual dictionary methods are supported. In addition, there is a maps attribute, a method for creating new subcontexts, and a
property for accessing all but the first mapping:

**maps**

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

**new_child**(m=None)

Returns a new ChainMap containing a new map followed by all of the maps in the current instance. If m is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to d.new_child() is equivalent to: ChainMap({}, *d.maps). This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

*Changed in version 3.4:* The optional m parameter was added.

**parents**

Property returning a new ChainMap containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the nonlocal keyword used in nested scopes. The use cases also parallel those for the built-in super() function. A reference to d.parents is equivalent to: ChainMap(*d.maps[1:]).

Note, the iteration order of a ChainMap() is determined by scanning the mappings last to first:

```python
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
```
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']

This gives the same ordering as a series of `dict.update()` calls starting with the last mapping:

```python
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

See also:

- The **MultiContext class** in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django's **Context class** for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the `new_child()` method and the `parents` property.
- The **Nested Contexts recipe** has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A **greatly simplified read-only version of Chainmap**.

**ChainMap Examples and Recipes**

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```python
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```
Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```python
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace)}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the `ChainMap` class to simulate nested contexts:

```python
c = ChainMap()       # Create root context
d = c.new_child()    # Create nested child context
e = c.new_child()    # Child of c, independent from
e.maps[0]            # Current context dictionary -
e.maps[-1]           # Root context -- like Python's
e.parents            # Enclosing context chain -- like

d['x'] = 1           # Set value in current context
d['x']               # Get first key in the chain of
del d['x']           # Delete from current context
list(d)              # All nested values
k in d               # Check all nested values
len(d)               # Number of nested values
```
The `ChainMap` class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```python
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

def __setitem__(self, key, value):
    for mapping in self.maps:
        if key in mapping:
            mapping[key] = value
        return
    self.maps[0][key] = value

def __delitem__(self, key):
    for mapping in self.maps:
        if key in mapping:
            del mapping[key]
        return
    raise KeyError(key)
```

```python
>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'white'}
>>> d['lion'] = 'orange'   # update an existing key
>>> d['snake'] = 'red'     # new keys get added
>>> del d['elephant']     # remove an existing key
>>> d                   # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {},)
```
**Counter objects**

A counter tool is provided to support convenient and rapid tallies. For example:

```python
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', ...
...    cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

```python
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 666), ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471)]
```

class collections.Counter([iterable-or-mapping])

A `Counter` is a `dict` subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The `Counter` class is similar to bags or multisets in other languages.

Elements are counted from an `iterable` or initialized from another `mapping` (or counter):

```python
>>> c = Counter()    # a new, empty counter
>>> c = Counter('gallahad')    # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})    # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)    # a new counter from keyword args
```
Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a `KeyError`:

```python
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```python
>>> c['sausage'] = 0
# count of a missing element is zero
>>> del c['sausage']
# del actually removes the entry
```

*New in version 3.1.*

Counter objects support three methods beyond those available for all dictionaries:

**elements()**

Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, `elements()` will ignore it.

```python
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

**most_common([n])**

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, `most_common()` returns all elements in the counter. Elements
with equal counts are ordered arbitrarily:

```python
>>> Counter('abracadabra').most_common(3)  # doctest:
[('a', 5), ('r', 2), ('b', 2)]
```

**subtract([iterable-or-mapping])**

Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like `dict.update()` but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```python
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

*New in version 3.2.*

The usual dictionary methods are available for *Counter* objects except for two which work differently for counters.

**fromkeys(iterable)**

This class method is not implemented for *Counter* objects.

**update([iterable-or-mapping])**

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like `dict.update()` but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of *(key, value)* pairs.
Common patterns for working with `Counter` objects:

```python
total of all counts
reset all counts
list unique elements
list unique elements
convert to a set
clear
convert to a regular dictionary
items
convert from a list
convert from a list
n least common elements
remove zero and negative counts
```

Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```python
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d
# add two counters together:
Counter({'a': 4, 'b': 3})
>>> c - d
# subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d
# intersection: min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d
# union: max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})

_New in version 3.3:_ Added support for unary plus, unary minus, and in-place multiset operations.

**Note:** Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The `Counter` class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you _could_ store anything in the value field.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The `elements()` method requires integer counts. It ignores zero and negative counts.
See also:

- **Bag class** in Smalltalk.
- Wikipedia entry for **Multisets**.
- **C++ multisets** tutorial with examples.
- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.
- To enumerate all distinct multisets of a given size over a given set of elements, see `itertools.combinations_with_replacement()`:

```
map(Counter, combinations_with_replacement('ABC'))
```
deque objects

class collections.deque([iterable[, maxlen]])

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appendes and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

If maxlen is not specified or is None, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(x)
Add $x$ to the right side of the deque.

**appendleft**($x$)
Add $x$ to the left side of the deque.

**clear**()
Remove all elements from the deque leaving it with length 0.

**copy**()
Create a shallow copy of the deque.

*New in version 3.5.*

**count**($x$)
Count the number of deque elements equal to $x$.

*New in version 3.2.*

**extend**(*iterable*)
Extend the right side of the deque by appending elements from the iterable argument.

**extendleft**(*iterable*)
Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

**index**($x$, [start[, stop]])
Return the position of $x$ in the deque (at or after index *start* and before index *stop*). Returns the first match or raises *ValueError* if not found.

*New in version 3.5.*
**insert**(\(i, x\))
Insert \(x\) into the deque at position \(i\).

If the insertion would cause a bounded deque to grow beyond \(maxlen\), an **IndexError** is raised.

*New in version 3.5.*

**pop()**
Remove and return an element from the right side of the deque. If no elements are present, raises an **IndexError**.

**popleft()**
Remove and return an element from the left side of the deque. If no elements are present, raises an **IndexError**.

**remove**(\(value\))
Remove the first occurrence of \(value\). If not found, raises a **ValueError**.

**reverse()**
Reverse the elements of the deque in-place and then return **None**.

*New in version 3.2.*

**rotate**\((n=1)\)
Rotate the deque \(n\) steps to the right. If \(n\) is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to \(d.appendleft(d.pop())\), and rotating one step to the left is equivalent to \(d.append(d.popleft())\).
Deque objects also provide one read-only attribute:

**maxlen**

Maximum size of a deque or None if unbounded.

*New in version 3.1.*

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[-1]`. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`. Example:

```python
>>> from collections import deque
>>> d = deque('ghi')  # make a new deque with three items
>>> for elem in d:  # iterate over the deque's elements
...     print(elem.upper())
G
H
I
>>> d.append('j')  # add a new entry to the right side
>>> d.appendleft('f')  # add a new entry to the left side
>>> d  # show the representation
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop()  # return and remove the rightmost item
'j'
>>> d.popleft()  # return and remove the leftmost item
'f'
```
deque Recipes

```python
>>> list(d)  # list the contents
['g', 'h', 'i']
>>> d[0]    # peek at leftmost item
'g'
>>> d[-1]   # peek at rightmost item
'i'

>>> list(reversed(d))  # list the contents in reverse
['i', 'h', 'g']

>>> 'h' in d  # search the deque
True

>>> d.extend('jkl')  # add multiple elements
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> d.rotate(1)     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])

>>> d.rotate(-1)    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))  # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])

>>> d.clear()        # empty the deque

>>> d.pop()          # cannot pop from an empty deque

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')  # extendleft()

>>> d
deque(['c', 'b', 'a'])
```
This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```python
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```python
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 4
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler can be implemented with input iterators stored in a `deque`. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with `popleft()`; otherwise, it can be cycled back to the end with the `rotate()` method:

```python
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
```
iterators = deque(map(iter, iterables))
while iterators:
    try:
        while True:
            yield next(iterators[0])
            iterators.rotate(-1)
    except StopIteration:
        # Remove an exhausted iterator.
        iterators.popleft()
```

The `rotate()` method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement deque slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`. 
**defaultdict objects**

```python
class collections.defaultdict([default_factory[,...]])
```

Returns a new dictionary-like object. `defaultdict` is a subclass of the built-in `dict` class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the `dict` class and is not documented here.

The first argument provides the initial value for the `default_factory` attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the `dict` constructor, including keyword arguments.

`defaultdict` objects support the following method in addition to the standard `dict` operations:

```python
__missing__(key)
```

If the `default_factory` attribute is `None`, this raises a `KeyError` exception with the `key` as argument.

If `default_factory` is not `None`, it is called without arguments to provide a default value for the given `key`, this value is inserted in the dictionary for the `key`, and returned.

If calling `default_factory` raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the `dict` class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.
Note that `__missing__()` is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using `default_factory`.

`defaultdict` objects support the following instance variable:

**`default_factory`**

This attribute is used by the `__missing__()` method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

**defaultdict** Examples

Using `list` as the `default_factory`, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```python
glp> s = [('yellow', 1), ('blue', 2), ('yellow', 3),
     >>> d = defaultdict(list)
     >>> for k, v in s:
     ...     d[k].append(v)
     ...
     >>> sorted(d.items())
     [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than
an equivalent technique using `dict.setdefault()`:

```python
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to `int` makes the `defaultdict` useful for counting (like a bag or multiset in other languages):

```python
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls `int()` to supply a default count of zero. The increment operation then builds up the count for each letter.

The function `int()` which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```python
>>> def constant_factory(value):
...     return lambda: value

>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
```
Setting the `default_factory` to `set` makes the `defaultdict` useful for building a dictionary of sets:

```python
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```
namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

`collections.namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)`

Returns a new tuple subclass named `typename`. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The `field_names` are a sequence of strings such as `['x', 'y']`. Alternatively, `field_names` can be a single string with each fieldname separated by whitespace and/or commas, for example `x y` or `x, y`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as `class`, `for`, `return`, `global`, `pass`, or `raise`.

If `rename` is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`. 
eliminating the keyword `def` and the duplicate fieldname `abc`.

`defaults` can be None or an iterable of default values. Since fields with a default value must come after any fields without a default, the `defaults` are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to 1, and `z` will default to 2.

If `module` is defined, the `__module__` attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

*Changed in version 3.1:* Added support for `rename`.

*Changed in version 3.6:* The `verbose` and `rename` parameters became keyword-only arguments.

*Changed in version 3.6:* Added the `module` parameter.

*Changed in version 3.7:* Remove the `verbose` parameter and the `_source` attribute.

*Changed in version 3.7:* Added the `defaults` parameter and the `_field_defaults` attribute.

```python
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)  # instantiate with positional arguments
>>> p[0] + p[1]  # indexable like the plain tuple (11, 22)
33
```
Named tuples are especially useful for assigning field names to result tuples returned by the `csv` or `sqlite3` modules:

```python
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open('employee_data.csv'))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall())�
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

**classmethod** `somenamedtuple._make(iterable)`

Class method that makes a new instance from an existing sequence or iterable.

```python
>>> t = [11, 22]
>>> Point._make(t)
```
Point(x=11, y=22)
somenamedtuple._asdict()
Return a new dict which maps field names to their corresponding values:

```python
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

*Changed in version 3.1:* Returns an OrderedDict instead of a regular dict.
somenamedtuple._replace(**kwargs)
Return a new instance of the named tuple replacing specified fields with new values:

```python
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)
```

```python
>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=25)
```
somenamedtuple._fields
Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```python
>>> p._fields
('x', 'y')
```

```python
>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color)
>>> Pixel(11, 22, 128, 255, 0)
```
somenamedtuple._fields_defaults
Dictionary mapping field names to default values.

```python
>>> Account = namedtuple('Account', ['type', 'balance'])
>>> Account._fields_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the `getattr()` function:

```python
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in Unpacking Argument Lists):

```python
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```python
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
```
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x=3.000  y=4.000  hypot=5.000
Point: x=14.000  y=0.714  hypot=14.018

The subclass shown above sets __slots__ to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

>>> Point3D = namedtuple('Point3D', Point._fields +
...
Docstrings can be customized by making direct assignments to the __doc__ fields:

>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'

*Changed in version 3.5:* Property docstrings became writeable.

Default values can be implemented by using _replace() to customize a prototype instance:

>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
See also:

- Recipe for named tuple abstract base class with a metaclass mix-in by Jan Kaliszewski. Besides providing an abstract base class for named tuples, it also supports an alternate metaclass-based constructor that is convenient for use cases where named tuples are being subclassed.
- See types.SimpleNamespace() for a mutable namespace based on an underlying dictionary instead of a tuple.
- See typing.NamedTuple() for a way to add type hints for named tuples.
OrderedDict objects

Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in dict class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

Some differences from dict still remain:

- The regular dict was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The OrderedDict was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, OrderedDict can handle frequent reordering operations better than dict. This makes it suitable for tracking recent accesses (for example in an LRU cache).
- The equality operation for OrderedDict checks for matching order.
- The popitem() method of OrderedDict has a different signature. It accepts an optional argument to specify which item is popped.
- OrderedDict has a move_to_end() method to efficiently reposition an element to an endpoint.
- Until Python 3.8, dict lacked a __reversed__() method.

class collections.OrderedDict([items])

Return an instance of a dict subclass that has methods specialized for rearranging dictionary order.

New in version 3.1.
**popitem**(last=True)

The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in **LIFO** order if *last* is true or **FIFO** order if false.

**move_to_end**(key, last=True)

Move an existing key to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the key does not exist:

```python
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

*New in version 3.2.*

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as *list(od1.items())==list(od2.items()*). Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

*Changed in version 3.5:* The items, keys, and values views of *OrderedDict* now support reverse iteration using *reversed()*.
Changed in version 3.6: With the acceptance of PEP 468, order is retained for keyword arguments passed to the OrderedDict constructor and its update() method.

**OrderedDict Examples and Recipes**

It is straightforward to create an ordered dictionary variant that remembers the order the keys were last inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```python
class LastUpdatedOrderedDict(OrderedDict):
    "Store items in the order the keys were last added"

def __setitem__(self, key, value):
    super().__setitem__(key, value)
    super().move_to_end(key)
```

An OrderedDict would also be useful for implementing variants of functools.lru_cache():

```python
class LRU(OrderedDict):
    "Limit size, evicting the least recently looked-up key when full"

def __init__(self, maxsize=128, *args, **kwds):
    self.maxsize = maxsize
    super().__init__(*args, **kwds)

    def __getitem__(self, key):
        value = super().__getitem__(key)
        self.move_to_end(key)
        return value
```
def __setitem__(self, key, value):
    super().__setitem__(key, value)
    if len(self) > self.maxsize:
        oldest = next(iter(self))
        del self[oldest]
**UserDict objects**

The class, `UserDict` acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from `dict`; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

```python
class collections/UserDict([initialdata])
```

Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the `data` attribute of `UserDict` instances. If `initialdata` is provided, `data` is initialized with its contents; note that a reference to `initialdata` will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings, `UserDict` instances provide the following attribute:

**data**

A real dictionary used to store the contents of the `UserDict` class.
**UserList objects**

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from `list`; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.**UserList**([`list`])  
Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the `data` attribute of `UserList` instances. The instance's contents are initially set to a copy of `list`, defaulting to the empty list `[]`. `list` can be any iterable, for example a real Python list or a `UserList` object.

In addition to supporting the methods and operations of mutable sequences, `UserList` instances provide the following attribute:

**data**

A real `list` object used to store the contents of the `UserList` class.

**Subclassing requirements:** Subclasses of `UserList` are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.
If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.
UserString objects

The class, UserString acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from str; however, this class can be easier to work with because the underlying string is accessible as an attribute.

```python
class collections.UserString(seq)
```

Class that simulates a string object. The instance’s content is kept in a regular string object, which is accessible via the data attribute of UserString instances. The instance's contents are initially set to a copy of seq. The seq argument can be any object which can be converted into a string using the built-in str() function.

In addition to supporting the methods and operations of strings, UserString instances provide the following attribute:

```python
data
```

A real str object used to store the contents of the UserString class.

Changed in version 3.5: New methods __getnewargs__, __rmod__, casefold, format_map, isprintable, and maketrans.
collections.abc — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the collections module.

Source code: Lib/_collections_abc.py

This module provides abstract base classes that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.
The collections module offers the following ABCs:

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Methods</th>
<th>Mixin Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td></td>
<td><strong>contains</strong></td>
<td></td>
</tr>
<tr>
<td>Hashable</td>
<td></td>
<td><strong>hash</strong></td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Iterable</td>
<td><strong>next</strong>, <strong>iter</strong></td>
<td>close <strong>iter</strong></td>
</tr>
<tr>
<td>Reversible</td>
<td>Iterable</td>
<td><strong>reversed</strong></td>
<td></td>
</tr>
<tr>
<td>Generator</td>
<td>Iterator</td>
<td>send, throw</td>
<td>close <strong>iter</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>next</strong>, <strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td><strong>call</strong></td>
<td></td>
</tr>
<tr>
<td>Collection</td>
<td>Sized, Iterable, Container</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Reversible, Collection</td>
<td><strong>getitem</strong>, <strong>len</strong></td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
</tr>
<tr>
<td>MutableSequence</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>len</strong></td>
<td>Inherited Seque method, append rever</td>
</tr>
<tr>
<td>ByteString</td>
<td>Sequence</td>
<td>insert</td>
<td>extend</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remove</td>
<td><strong>iadd</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Collection</th>
<th><strong>contains</strong>, <strong>getitem</strong>, <strong>len</strong></th>
<th>Inherited Sequence methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>le</strong>, <strong>lt</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong>, <strong>and</strong>, <strong>or</strong>, <strong>sub</strong>, <strong>xor</strong>, isdisjoint</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MutableSet</th>
<th>Set</th>
<th><strong>contains</strong>, <strong>getitem</strong>, <strong>iter</strong>, <strong>len</strong>, add, discard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Inherited method clear, remove, <strong>ior</strong>, <strong>iand</strong>, <strong>ixor</strong>, <strong>isub</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Collection</th>
<th><strong>getitem</strong>, <strong>iter</strong>, <strong>len</strong></th>
<th>Inherited Mapping methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>contains</strong>, keys, values, <strong>eq</strong>, <strong>ne</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MutableMapping</th>
<th>Mapping</th>
<th><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>,</th>
<th>Inherited Mapping methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>pop, p</td>
<td></td>
</tr>
</tbody>
</table>
### class collections.abc.

- **Container**
- **Hashable**
- **Sized**
- **Callable**

ABCs for classes that provide respectively the methods `__contains__()`, `__hash__()`, `__len__()`, and `__call__()`.  

### class collections.abc.

- **Iterable**

ABC for classes that provide the `__iter__()` method.  

Checking `isinstance(obj, Iterable)` detects classes that...
are registered as `Iterable` or that have an `__iter__()` method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is `iterable` is to call `iter(obj)`.

```python
class collections.abc.Collection
    ABC for sized iterable container classes.

    New in version 3.6.

class collections.abc.Iterator
    ABC for classes that provide the `__iter__()` and `__next__()` methods. See also the definition of `iterator`.

class collections.abc.Reversible
    ABC for iterable classes that also provide the `__reversed__()` method.

    New in version 3.6.

class collections.abc.Generator
    ABC for generator classes that implement the protocol defined in PEP 342 that extends iterators with the `send()`, `throw()` and `close()` methods. See also the definition of `generator`.

    New in version 3.5.

class collections.abc.Sequence
class collections.abc.MutableSequence
class collections.abc.ByteString
    ABCs for read-only and mutable sequences.

    Implementation note: Some of the mixin methods, such as
__iter__(), __reversed__() and index(), make repeated calls to the underlying __getitem__() method. Consequently, if __getitem__() is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

*Changed in version 3.5:* The index() method added support for *stop* and *start* arguments.

class collections.abc.Set
class collections.abc.MutableSet

ABCs for read-only and mutable sets.

class collections.abc.Mapping
class collections.abc.MutableMapping

ABCs for read-only and mutable mappings.

class collections.abc.MappingView
class collections.abc.ItemsView
class collections.abc.KeysView
class collections.abc.ValuesView

ABCs for mapping, items, keys, and values views.

class collections.abc.Awaitable

ABC for awaitable objects, which can be used in await expressions. Custom implementations must provide the __await__() method.

Coroutine objects and instances of the Coroutine ABC are all instances of this ABC.
In CPython, generator-based coroutines (generators decorated with `types.coroutine()` or `asyncio.coroutine()`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

*New in version 3.5.*

```python
class collections.abc.Coroutine

ABC for coroutine compatible classes. These implement the following methods, defined in *Coroutine Objects*: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All `Coroutine` instances are also instances of `Awaitable`. See also the definition of `coroutine`.
```

*Note:* In CPython, generator-based coroutines (generators decorated with `types.coroutine()` or `asyncio.coroutine()`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

*New in version 3.5.*

```python
class collections.abc.AsyncIterable

ABC for classes that provide `__aiter__` method. See also the definition of `asynchronous iterable`.
```

*New in version 3.5.*

```python
class collections.abc.AsyncIterator
```
ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of asynchronous iterator.

*New in version 3.5.*

class collections.abc.AsyncGenerator

ABC for asynchronous generator classes that implement the protocol defined in PEP 525 and PEP 492.

*New in version 3.6.*

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full Set API, it is only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)
```
def __iter__(self):
    return iter(self.elements)

def __contains__(self, value):
    return value in self.elements

def __len__(self):
    return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2  # The __and__() method

Notes on using Set and MutableSet as a mixin:

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form ClassName(iterable). That assumption is factored-out to an internal classmethod called _from_iterable() which calls cls(iterable) to produce a new set. If the Set mixin is being used in a class with a different constructor signature, you will need to override _from_iterable() with a classmethod that can construct new instances from an iterable argument.

2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine __le__() and __ge__(), then the other operations will automatically follow suit.

3. The Set mixin provides a _hash() method to compute a hash value for the set; however, __hash__() is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both Set() and Hashable(), then define __hash__ = Set._hash.

See also:
- **OrderedSet recipe** for an example built on **MutableSet**.
- For more about ABCs, see the **abc** module and **PEP 3119**.
This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all `k`, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0].`

The API below differs from textbook heap algorithms in two aspects:
(a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing.
(b) Our `pop` method returns the smallest item, not the largest (called a “min heap” in textbooks; a “max heap” is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function `heapify()`.

The following functions are provided:
heapq. **heappush**(heap, item)

Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq. **heappop**(heap)

Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, **IndexError** is raised. To access the smallest item without popping it, use `heap[0]`.

heapq. **heappushpop**(heap, item)

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

heapq. **heapify**(x)

Transform list *x* into a heap, in-place, in linear time.

heapq. **heapreplace**(heap, item)

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn’t change. If the heap is empty, **IndexError** is raised.

This one step operation is more efficient than a `heappop()` followed by `heappush()` and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn’t desired, consider using `heappushpop()` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.
heapq.**merge**(*iterables, key=None, reverse=False)

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an iterator over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

`key` specifies a **key function** of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

`reverse` is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

*Changed in version 3.5:* Added the optional `key` and `reverse` parameters.

heapq.**nlargest**(*n, iterable, key=None*)

Return a list with the *n* largest elements from the dataset defined by `iterable`. `key`, if provided, specifies a function of one argument that is used to extract a comparison key from each element in `iterable` (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[::-n]`. 
heapq.nsmallest($n$, iterable, key=None)

Return a list with the $n$ smallest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in iterable (for example, key=str.lower). Equivalent to: sorted(iterable, key=key)[:-n].

The latter two functions perform best for smaller values of $n$. For larger values, it is more efficient to use the sorted() function. Also, when $n==1$, it is more efficient to use the built-in min() and max() functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.
Basic Examples

A `heapsort` can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```python
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]

>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike `sorted()`, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```python
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```
Priority Queue Implementation Notes

A **priority queue** is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```python
def from dataclasses import dataclass, field
def from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```
The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```python
pq = []  # list of entries arranged in a heap
def add_task(task, priority=0):
    # Add a new task or update the priority of an existing task.
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    # Mark an existing task as REMOVED. Raise KeyError if not found.
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    # Remove and return the lowest priority task. Raise KeyError if empty.
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

```python
entry_finder = {}  # mapping of tasks to entries
REMOVED = '<removed-task>'  # placeholder for a removed task
counter = itertools.count()  # unique sequence counter
```
Theory

Heaps are arrays for which \( a[k] \leq a[2k+1] \) and \( a[k] \leq a[2k+2] \) for all \( k \), counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that \( a[0] \) is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are \( k \), not \( a[k] \):

![Heap Diagram]

In the tree above, each cell \( k \) is topping \( 2k+1 \) and \( 2k+2 \). In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell “wins” over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the
overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not “better” than the last 0’th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised [1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you’ll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.
Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a ‘heap’ module around. :-)
bisect — Array bisection algorithm

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called bisect because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

bisect.\texttt{bisect\_left}(a, x, lo=0, hi=len(a))

Locate the insertion point for \(x\) in \(a\) to maintain sorted order. The parameters \(lo\) and \(hi\) may be used to specify a subset of the list which should be considered; by default the entire list is used. If \(x\) is already present in \(a\), the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to \texttt{list.insert()} assuming that \(a\) is already sorted.

The returned insertion point \(i\) partitions the array \(a\) into two halves so that \(\text{all}(\text{val} < x \text{ for val in a[lo:i]})\) for the left side and \(\text{all}(\text{val} \geq x \text{ for val in a[i:hi]})\) for the right side.

\begin{align*}
\text{bisect.\texttt{bisect\_right}}(a, x, lo=0, hi=len(a)) \\
\text{bisect.\texttt{bisect}}(a, x, lo=0, hi=len(a))
\end{align*}

Similar to \texttt{bisect\_left()}, but returns an insertion point which
comes after (to the right of) any existing entries of \( x \) in \( a \).

The returned insertion point \( i \) partitions the array \( a \) into two halves so that \( \text{all}(\text{val} \leq x \text{ for val in a[lo:i]}) \) for the left side and \( \text{all}(\text{val} > x \text{ for val in a[i:hi]}) \) for the right side.

\[
bisect.\text{insort\_left}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]

Insert \( x \) in \( a \) in sorted order. This is equivalent to
\[
a.\text{insert}(\text{bisect.bisect\_left}(a, x, \text{lo}, \text{hi}), x)
\]
assuming that \( a \) is already sorted. Keep in mind that the \( O(\log n) \) search is dominated by the slow \( O(n) \) insertion step.

\[
bisect.\text{insort\_right}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]
\[
bisect.\text{insort}(a, x, \text{lo}=0, \text{hi}=\text{len}(a))
\]

Similar to \( \text{insort\_left()} \), but inserting \( x \) in \( a \) after any existing entries of \( x \).

**See also:** [SortedCollection recipe](https://docs.python.org/3/library/bisect.html) that uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.
Searching Sorted Lists

The above `bisect()` functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```python
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```
def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
Other Examples

The `bisect()` function can be useful for numeric table lookups. This example uses `bisect()` to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an ‘A’, 80 to 89 is a ‘B’, and so on:

```python
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades=['F', 'D', 'C', 'B', 'A']):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Unlike the `sorted()` function, it does not make sense for the `bisect()` functions to have `key` or `reversed` arguments because that would lead to an inefficient design (successive calls to bisect functions would not “remember” all of the previous key lookups).

Instead, it is better to search a list of precomputed keys to find the index of the record in question:

```python
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
```
```python
>>> keys = [r[1] for r in data]  # precompute list of keys
>>> keys
[0, 1, 5, 8]
```
```python
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```
array — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

<table>
<thead>
<tr>
<th>Type code</th>
<th>C Type</th>
<th>Python Type</th>
<th>Minimum size in bytes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>signed char</td>
<td>int</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>unsigned char</td>
<td>int</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'u'</td>
<td>Py_UNICODE</td>
<td>Unicode character</td>
<td>2</td>
<td>(1)</td>
</tr>
<tr>
<td>'h'</td>
<td>signed short</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned short</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>signed int</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned int</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td>signed long</td>
<td>int</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'L'</td>
<td>unsigned long</td>
<td>int</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'q'</td>
<td>signed long</td>
<td>int</td>
<td>8</td>
<td>(2)</td>
</tr>
<tr>
<td>'Q'</td>
<td>unsigned long</td>
<td>int</td>
<td>8</td>
<td>(2)</td>
</tr>
</tbody>
</table>
Notes:

1. The 'u' type code corresponds to Python's obsolete unicode character (Py_UNICODE which is wchar_t). Depending on the platform, it can be 16 bits or 32 bits.

   'u' will be removed together with the rest of the Py_UNICODE API.

   *Deprecated since version 3.3, will be removed in version 4.0.*

2. The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long, or, on Windows, __int64.

   *New in version 3.3.*

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute.

The module defines the following type:

```python
class array.array(typecode[, initializer])
```

A new array whose items are restricted by `typecode`, and initialized from the optional `initializer` value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

If given a list or string, the initializer is passed to the new array's `fromlist()`, `frombytes()`, or `fromunicode()` method (see
below) to add initial items to the array. Otherwise, the iterable initializer is passed to the `extend()` method.

`array.typecodes`  
A string with all available type codes.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, `TypeError` is raised. Array objects also implement the buffer interface, and may be used wherever bytes-like objects are supported.

The following data items and methods are also supported:

`array.typecode`  
The typecode character used to create the array.

`array.itemsize`  
The length in bytes of one array item in the internal representation.

`array.append(x)`  
Append a new item with value x to the end of the array.

`array.buffer_info()`  
Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations
are applied to it.

**Note:** When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in Buffer Protocol.

**array.byteswap()**

“Byteswap” all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, `RuntimeError` is raised. It is useful when reading data from a file written on a machine with a different byte order.

**array.count(x)**

Return the number of occurrences of x in the array.

**array.extend(iterable)**

Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, `TypeError` will be raised. If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array.

**array.frombytes(s)**

Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the `fromfile()` method).

*New in version 3.2:* `fromstring()` is renamed to `frombytes()` for clarity.
array. **fromfile**(*f*, *n*)
Read *n* items (as machine values) from the file object *f* and append them to the end of the array. If less than *n* items are available, **EOFError** is raised, but the items that were available are still inserted into the array. *f* must be a real built-in file object; something else with a read() method won’t do.

array. **fromlist**(*list*)
Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

array. **fromstring**()
Deprecated alias for **frombytes()**.

array. **fromunicode**(*s*)
Extends this array with data from the given unicode string. The array must be a type 'u' array; otherwise a **ValueError** is raised. Use `array.frombytes(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

array. **index**(*x*)
Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array.

array. **insert**(*i*, *x*)
Insert a new item with value *x* in the array before position *i*. Negative values are treated as being relative to the end of the array.

array. **pop**(*[i]*)
Removes the item with the index \( i \) from the array and returns it. The optional argument defaults to \(-1\), so that by default the last item is removed and returned.

\[ \text{array.} \texttt{remove}(x) \]
Remove the first occurrence of \( x \) from the array.

\[ \text{array.} \texttt{reverse}() \]
Reverse the order of the items in the array.

\[ \text{array.} \texttt{tobytes}() \]
Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the \texttt{tofile()} method.)

*New in version 3.2:* \texttt{tostring()} is renamed to \texttt{tobytes()} for clarity.

\[ \text{array.} \texttt{tofile}(f) \]
Write all items (as machine values) to the file object \( f \).

\[ \text{array.} \texttt{tolist}() \]
Convert the array to an ordinary list with the same items.

\[ \text{array.} \texttt{tostring}() \]
Deprecated alias for \texttt{tobytes()}.

\[ \text{array.} \texttt{tounicode}() \]
Convert the array to a unicode string. The array must be a type \'u\' array; otherwise a \texttt{ValueError} is raised. Use \texttt{array.tobytes().decode(enc)} to obtain a unicode string from an array of some other type.
When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The `initializer` is omitted if the array is empty, otherwise it is a string if the `typecode` is 'u', otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using `eval()`, so long as the `array` class has been imported using `from array import array`. Examples:

```python
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

**See also:**

**Module** `struct`  
Packing and unpacking of heterogeneous binary data.

**Module** `xdrlib`  
Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

**The Numerical Python Documentation**  
The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it’s desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don’t keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to
that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

**WeakKeyDictionary** and **WeakValueDictionary** use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. **WeakSet** implements the **set** interface, but keeps weak references to its elements, just like a **WeakKeyDictionary** does.

**finalize** provides a straightforward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or **finalize** is all they need – it’s not usually necessary to create your own weak references directly. The low-level machinery is exposed by the **weakref** module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some file objects, generators, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

*Changed in version 3.2:* Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as **list** and **dict** do not directly support weak references but can add support through subclassing:
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referenceable

Other built-in types such as tuple and int do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see Weak Reference Support.

class weakref.ref(object[, callback])

    Return a weak reference to object. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause None to be returned. If callback is provided and not None, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

    It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

    Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object’s __del__() method.
Weak references are hashable if the `object` is hashable. They will maintain their hash value even after the `object` was deleted. If `hash()` is called the first time only after the `object` was deleted, the call will raise `TypeError`.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the `callback`). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__`__callback__

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

*Changed in version 3.4: Added the `__callback__` attribute.*

`weakref.proxy(object[, callback])`

Return a proxy to `object` which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether `object` is callable. Proxy objects are not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. `callback` is the same as the parameter of the same name to the `ref()` function.
**weakref.getweakrefcount**(object)

Return the number of weak references and proxies which refer to object.

**weakref.getweakrefs**(object)

Return a list of all weak reference and proxy objects which refer to object.

**class weakref.WeakKeyDictionary([dict])**

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

**Note:** Caution: Because a *WeakKeyDictionary* is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a *WeakKeyDictionary* because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

*WeakKeyDictionary* objects have an additional method that exposes the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

**WeakKeyDictionary.keyrefs()**

Return an iterable of the weak references to the keys.
class weakref.WeakValueDictionary([dict])

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Note: Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

WeakValueDictionary objects have an additional method that has the same issues as the keyrefs() method of WeakKeyDictionary objects.

WeakValueDictionary.valuerefs()

Return an iterable of the weak references to the values.

class weakref.WeakSet([elements])

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class weakref.WeakMethod(method)

A custom ref subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. WeakMethod has special code to recreate the bound method until either the object or the original function dies:

>>> class C:
New in version 3.4.

class weakref.\texttt{finalize}(obj, func, *args, **kwargs)

Return a callable finalizer object which will be called when \texttt{obj} is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered \textit{alive} until it is called (either explicitly or at garbage collection), and after that it is \textit{dead}. Calling a live finalizer returns the result of evaluating \texttt{func(*arg, **kwargs)}, whereas calling a dead finalizer returns \texttt{None}.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's \texttt{__del__()} method or a weak reference's
callback.

When the program exits, each remaining live finalizer is called unless its `atexit` attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the interpreter shutdown when module globals are liable to have been replaced by `None`.

```python
__call__(self)

If `self` is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If `self` is dead then return `None`.

detach(self)

If `self` is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If `self` is dead then return `None`.

peek(self)

If `self` is alive then return the tuple `(obj, func, args, kwargs)`. If `self` is dead then return `None`.

alive

Property which is true if the finalizer is alive, false otherwise.

atexit

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which `atexit` is true. They are called in reverse order of creation.

Note: It is important to ensure that `func`, `args` and `kwargs` do not own any references to `obj`, either directly or indirectly, since
otherwise obj will never be garbage collected. In particular, func should not be a bound method of obj.

New in version 3.4.

weakref. ReferenceType
   The type object for weak references objects.

weakref. ProxyType
   The type object for proxies of objects which are not callable.

weakref. CallableProxyType
   The type object for proxies of callable objects.

weakref. ProxyTypes
   Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

exception weakref. ReferenceError
   Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard ReferenceError exception.

See also:

PEP 205 - Weak References
   The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.
Weak Reference Objects

Weak reference objects have no methods and no attributes besides `ref.__callback__`. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```python
>>> import weakref
>>> class Object:
...     pass
...>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None`:

```python
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```python
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobn"
else:
    print("Object is still live!")
```
Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the `WeakValueDictionary` to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that’s returned when the referent is accessed:

```python
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of times the reference has been called."
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
```
self.__counter += 1
ob = (ob, self.__counter)
return ob
Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```python
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```
Finalizer Objects

The main benefit of using `finalize` is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```python
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```python
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f() # callback not called because finalizer dead
>>> del obj # callback not called because finalizer dead
```

You can unregister a finalizer using its `detach()` method. This kills
the finalizer and returns the arguments passed to the constructor when it was created.

```python
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
```

CALLBACK

Unless you set the `atexit` attribute to `False`, a finalizer will be called when the program exits if it is still alive. For instance

```python
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```
Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```python
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
        self.name = None

@property
def removed(self):
    return self.name is None

def __del__(self):
    self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during interpreter shutdown. So this
code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```python
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

@property
def removed(self):
    return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```python
import weakref, sys

def unloading_module():
    # implicit reference to the module globals from
    weakref.finalize(sys.modules[__name__], unloading_module)
```
**Note:** If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.
**types** — Dynamic type creation and names for built-in types

**Source code:** Lib/types.py

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like `int` or `str` are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.
Dynamic Type Creation

`types.new_class(name, bases=(), kwds=None, exec_body=None)`

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as metaclass).

The `exec_body` argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: ns`.

*New in version 3.3.*

`types.prepare_class(name, bases=(), kwds=None)`

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

`metaclass` is the appropriate metaclass, `namespace` is the prepared class namespace and `kwds` is an updated copy of the passed in `kwds` argument with any 'metaclass' entry removed.
If no \textit{kwds} argument is passed in, this will be an empty dict.

\textit{New in version 3.3.}

\textit{Changed in version 3.6}: The default value for the \texttt{namespace} element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a \underline{\texttt{__prepare__}}\ method.

\textbf{See also:}

\textbf{Metaclasses}

Full details of the class creation process supported by these functions

\textbf{PEP 3115 - Metaclasses in Python 3000}

Introduced the \underline{\texttt{__prepare__}} namespace hook

\texttt{types.resolve_bases(bases)}

Resolve MRO entries dynamically as specified by \textbf{PEP 560}.

This function looks for items in \textit{bases} that are not instances of \texttt{type}, and returns a tuple where each such object that has an \underline{\texttt{__mro_entries__}} method is replaced with an unpacked result of calling this method. If a \textit{bases} item is an instance of \texttt{type}, or it doesn't have an \underline{\texttt{__mro_entries__}} method, then it is included in the return tuple unchanged.

\textit{New in version 3.7.}

\textbf{See also:} \textbf{PEP 560} - Core support for typing module and generic types
Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for `isinstance()` or `issubclass()` checks.

Standard names are defined for the following types:

* `types.FunctionType`  
The type of user-defined functions and functions created by `lambda` expressions.

* `types.LambdaType`  
The type of `generator`-iterator objects, created by generator functions.

* `types.CoroutineType`  
The type of `coroutine` objects, created by `async def` functions.

  *New in version 3.5.*

* `types.AsyncGeneratorType`  
The type of asynchronous `generator`-iterator objects, created by asynchronous generator functions.

  *New in version 3.6.*

* `types.CodeType`
The type for code objects such as returned by `compile()`.

**types. MethodType**
The type of methods of user-defined class instances.

**types. BuiltinFunctionType**
**types. BuiltinMethodType**
The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term “built-in” means “written in C”.)

**types. WrapperDescriptorType**
The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

*New in version 3.7.*

**types. MethodWrapperType**
The type of **bound** methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

*New in version 3.7.*

**types. MethodDescriptorType**
The type of methods of some built-in data types such as `str.join()`.

*New in version 3.7.*

**types. ClassMethodDescriptorType**
The type of **unbound** class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

*New in version 3.7.*
class types. **ModuleType**(name, doc=None)

The type of modules. Constructor takes the name of the module to be created and optionally its docstring.

**Note:** Use `importlib.util.module_from_spec()` to create a new module if you wish to set the various import-controlled attributes.

__doc__
The docstring of the module. Defaults to None.

__loader__
The loader which loaded the module. Defaults to None.

*Changed in version 3.4:* Defaults to None. Previously the attribute was optional.

__name__
The name of the module.

__package__
Which package a module belongs to. If the module is top-level (i.e. not a part of any specific package) then the attribute should be set to '', else it should be set to the name of the package (which can be __name__ if the module is a package itself). Defaults to None.

*Changed in version 3.4:* Defaults to None. Previously the attribute was optional.

class types. **TracebackType**(tb_next, tb_frame, tb_lasti, tb_lineno)
The type of traceback objects such as found in `sys.exc_info()`
See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

**types.FrameType**
The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

See the language reference for details of the available attributes and operations.

**types.GetSetDescriptorType**
The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the `property` type, but for classes defined in extension modules.

**types.MemberDescriptorType**
The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the `property` type, but for classes defined in extension modules.

**CPython implementation detail:** In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

**class types.MappingProxyType(mapping)**
Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.
New in version 3.3.

**key in proxy**
Return True if the underlying mapping has a key key, else False.

**proxy[key]**
Return the item of the underlying mapping with key key. Raises a KeyError if key is not in the underlying mapping.

**iter(proxy)**
Return an iterator over the keys of the underlying mapping. This is a shortcut for iter(proxy.keys()).

**len(proxy)**
Return the number of items in the underlying mapping.

**copy()**
Return a shallow copy of the underlying mapping.

**get(key[, default])**
Return the value for key if key is in the underlying mapping, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

**items()**
Return a new view of the underlying mapping’s items ((key, value) pairs).

**keys()**
Return a new view of the underlying mapping’s keys.

**values()**
Return a new view of the underlying mapping's values.
Additional Utility Classes and Functions

class types.SimpleNamespace

A simple object subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike object, with SimpleNamespace you can add and remove attributes. If a SimpleNamespace object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code:

class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}\n".format(k, self.__dict__
        return "{}({})\n".format(type(self).__name_

    def __eq__(self, other):
        return self.__dict__ == other.__dict__

SimpleNamespace may be useful as a replacement for class NS: pass. However, for a structured record type use namedtuple() instead.

New in version 3.3.

types.DynamicClassAttribute(fget=None, fset=None, fdel=None, doc=None)
Route attribute access on a class to \_\_getattr\_.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class’s \_\_getattr\_ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see Enum for an example).

*New in version 3.4.*
Coroutine Utility Functions

types.coroutine(gen_func)

This function transforms a generator function into a coroutine function which returns a generator-based coroutine. The generator-based coroutine is still a generator iterator, but is also considered to be a coroutine object and is awaitable. However, it may not necessarily implement the __await__() method.

If gen_func is a generator function, it will be modified in-place.

If gen_func is not a generator function, it will be wrapped. If it returns an instance of collections.abc.Generator, the instance will be wrapped in an awaitable proxy object. All other types of objects will be returned as is.

New in version 3.5.
copy — Shallow and deep copy operations

Source code: Lib/copy.py

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

**copy**. `copy(x)`

Return a shallow copy of `x`.

**copy**. `deepcopy(x[, memo])`

Return a deep copy of `x`.

*exception* `copy.error`

Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the
Two problems often exist with deep copy operations that don’t exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The `deepcopy()` function avoids these problems by:

- keeping a memo dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does “copy” functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the `pickle` module.

Shallow copies of dictionaries can be made using `dict.copy()`, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module `pickle` for information on these methods. In fact, the `copy` module uses the registered pickle functions from the `copyreg` module.

In order for a class to define its own copy implementation, it can define special methods `__copy__() and __deepcopy__()`. The former is
called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the __deepcopy__() implementation needs to make a deep copy of a component, it should call the deepcopy() function with the component as first argument and the memo dictionary as second argument.

**See also:**

**Module pickle**

Discussion of the special methods used to support object state retrieval and restoration.
The `pprint` module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don’t fit within the allowed width. Construct `PrettyPrinter` objects explicitly if you need to adjust the width constraint.

Dictionaries are sorted by key before the display is computed.

The `pprint` module defines one class:

```python
class pprint.PrettyPrinter(indent=1, width=80, depth=None, stream=None, *, compact=False)
```

Construct a `PrettyPrinter` instance. This constructor understands several keyword parameters. An output stream may be set using the `stream` keyword; the only method used on the stream object is the file protocol’s `write()` method. If not specified, the `PrettyPrinter` adopts `sys.stdout`. The amount of indentation added for each recursive level is specified by `indent`; the default is one. Other values can cause output to look a little
odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by depth; if the data structure being printed is too deep, the next contained level is replaced by ... By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the width parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made. If compact is false (the default) each item of a long sequence will be formatted on a separate line. If compact is true, as many items as will fit within the width will be formatted on each output line.

Changed in version 3.4: Added the compact parameter.

```python
code_snippet
```
The **pprint** module also provides several shortcut functions:

```python
pprint.pformat(object, indent=1, width=80, depth=None, *, compact=False)
```

Return the formatted representation of `object` as a string. `indent`, `width`, `depth` and `compact` will be passed to the `PrettyPrinter` constructor as formatting parameters.

*Changed in version 3.4:* Added the `compact` parameter.

```python
pprint.pprint(object, stream=None, indent=1, width=80, depth=None, *, compact=False)
```

Prints the formatted representation of `object` on `stream`, followed by a newline. If `stream` is `None`, `sys.stdout` is used. This may be used in the interactive interpreter instead of the `print()` function for inspecting values (you can even reassign `print = pprint.pprint` for use within a scope). `indent`, `width`, `depth` and `compact` will be passed to the `PrettyPrinter` constructor as formatting parameters.

*Changed in version 3.4:* Added the `compact` parameter.

```python
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni'
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
]```
**pprint.** isreadable\( (object) \)**

Determine if the formatted representation of \( object \) is “readable,” or can be used to reconstruct the value using \( \texttt{eval()} \). This always returns \texttt{False} for recursive objects.

```python
>>> pprint.isreadable(stuff)
False
```

**pprint.** isrecursive\( (object) \)**

Determine if \( object \) requires a recursive representation.

One more support function is also defined:

**pprint.** saferepr\( (object) \)**

Return a string representation of \( object \), protected against recursive data structures. If the representation of \( object \) exposes a recursive entry, the recursive reference will be represented as \<Recursion on typename with id=number\>. The representation is not otherwise formatted.

```python
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs'
```
PrettyPrinter Objects

*PrettyPrinter* instances have the following methods:

**PrettyPrinter**. `pformat(object)`

Return the formatted representation of `object`. This takes into account the options passed to the *PrettyPrinter* constructor.

**PrettyPrinter**. `pprint(object)`

Print the formatted representation of `object` on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new *PrettyPrinter* objects don’t need to be created.

**PrettyPrinter**. `is_readable(object)`

Determine if the formatted representation of the object is “readable,” or can be used to reconstruct the value using `eval()`. Note that this returns `False` for recursive objects. If the *depth* parameter of the *PrettyPrinter* is set and the object is deeper than allowed, this returns `False`.

**PrettyPrinter**. `is_recursive(object)`

Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the `safepping()` implementation.

**PrettyPrinter**. `format(object, context, maxlevels, level)`
Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the *id()* of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be True. Recursive calls to the *format()* method should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be 0 if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.
Example

To demonstrate several uses of the `pprint()` function and its parameters, let’s fetch information about a project from PyPI:

```python
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

In its basic form, `pprint()` shows the whole object:

```python
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                'Intended Audience :: Developers',
                'License :: OSI Approved :: MIT License',
                'Programming Language :: Python :: 2',
                'Programming Language :: Python :: 2.6',
                'Programming Language :: Python :: 2.7',
                'Programming Language :: Python :: 3',
                'Programming Language :: Python :: 3.2',
                'Programming Language :: Python :: 3.3',
                'Programming Language :: Python :: 3.4',
                'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project

This is the description file for the

The file should use UTF-8 encoding
```
'ReStructured Text. It

will be used to generate the project

should be written for

that purpose.


Typical contents for this file would

the project, basic

usage examples, etc. Generally, includ

changelog in here is not

a good idea, although a simple "What

most recent version"

may be appropriate.',

description_content_type': None,

docs_url': None,

download_url': 'UNKnown',

downloads': \{'last_day': -1, \'last_month': -1, \'la

home_page': 'https://github.com/pypa/sampleproject

keywords': 'sample setuptools development',

license': 'MIT',

maintainer': None,

maintainer_email': None,

name': 'sampleproject',

package_url': 'https://pypi.org/project/sampleproj

platform': 'UNKnown',

project_url': 'https://pypi.org/project/sampleproj

project_urls': \{'Download': 'UNKNOWN',

    \'Homepage': 'https://github.com/pypa/samplepr

release_url': 'https://pypi.org/project/sampleproj

requires_dist': None,

requires_python': None,

summary': 'A sample Python project',

version': '1.2.0'}

The result can be limited to a certain depth (ellipsis is used for deeper contents):
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project

This is the description file for the project.
The file should use UTF-8 encoding and be written using ReStructured Text. It will be used to generate the project webpage on PyPI, and should be written for that purpose.

Typical contents for this file would include an overview of the project, basic usage examples, etc. Generally, including the changelog in here is not a good idea, although a simple "What's New" section for the most recent version may be appropriate.',
'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/
'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/
 'project_urls': {...},
Additionally, maximum character width can be suggested. If a long object cannot be split, the specified width will be exceeded:

```python
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project

=======================

This is the description file for the project.

The file should use UTF-8 encoding and be written using ReStructured Text. It will be used to generate the project webpage on PyPI, and should be written for that purpose.

Typical contents for this file would include an overview of the project, basic usage examples, etc. Generally, including the project changelog in here is not a good idea, although a simple "What's New" section for the most recent version may be appropriate.',
'description_content_type': None,
}
The `reprlib` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

```python
class reprlib.Repr
    Class which provides formatting services useful in implementing functions similar to the built-in `repr()`; size limits for different object types are added to avoid the generation of representations which are excessively long.

reprlib.aRepr
    This is an instance of `Repr` which is used to provide the `repr()` function described below. Changing the attributes of this object will affect the size limits used by `repr()` and the Python debugger.

reprlib.repr(obj)
    This is the `repr()` method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.
```

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a
@reprlib.recursive_repr(fillvalue="...")

Decorator for __repr__() methods to detect recursive calls within the same thread. If a recursive call is made, the fillvalue is returned, otherwise, the usual __repr__() call is made. For example:

```python
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self))
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<a'|'b'|'c'|...'|'x'>
```

New in version 3.2.
Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.$\texttt{maxlevel}$
Depth limit on the creation of recursive representations. The default is 6.

Repr.$\texttt{maxdict}$
Repr.$\texttt{maxlist}$
Repr.$\texttt{maxtuple}$
Repr.$\texttt{maxset}$
Repr.$\texttt{maxfrozenset}$
Repr.$\texttt{maxdeque}$
Repr.$\texttt{maxarray}$
Limits on the number of entries represented for the named object type. The default is 4 for $\texttt{maxdict}$, 5 for $\texttt{maxarray}$, and 6 for the others.

Repr.$\texttt{maxlong}$
Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is 40.

Repr.$\texttt{maxstring}$
Limit on the number of characters in the representation of the string. Note that the “normal” representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.
Repr. **maxother**

This limit is used to control the size of object types for which no specific formatting method is available on the Repr object. It is applied in a similar manner as maxstring. The default is 20.

Repr. **repr(obj)**

The equivalent to the built-in repr() that uses the formatting imposed by the instance.

Repr. **repr1(obj, level)**

Recursive implementation used by repr(). This uses the type of obj to determine which formatting method to call, passing it obj and level. The type-specific methods should call repr1() to perform recursive formatting, with level - 1 for the value of level in the recursive call.

Repr. **repr_TYPE(obj, level)**

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, TYPE is replaced by '_'.join(type(obj).__name__.split()). Dispatch to these methods is handled by repr1(). Type-specific methods which need to recursively format a value should call self.repr1(subobj, level - 1).
Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```python
import reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr'>:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))  # prints '<stdin>'
```
enum — Support for enumerations

New in version 3.4.

Source code: Lib/enum.py

An enumeration is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.
Module Contents

This module defines four enumeration classes that can be used to define unique sets of names and values: `Enum`, `IntEnum`, `Flag`, and `IntFlag`. It also defines one decorator, `unique()`, and one helper, `auto`.

```python
class enum.Enum
    Base class for creating enumerated constants. See section Functional API for an alternate construction syntax.

class enum.IntEnum
    Base class for creating enumerated constants that are also subclasses of `int`.

class enum.IntFlag
    Base class for creating enumerated constants that can be combined using the bitwise operators without losing their `IntFlag` membership. `IntFlag` members are also subclasses of `int`.

class enum.Flag
    Base class for creating enumerated constants that can be combined using the bitwise operations without losing their `Flag` membership.

enum.unique()
    Enum class decorator that ensures only one name is bound to any one value.

class enum.auto
    Instances are replaced with an appropriate value for Enum members.
```
New in version 3.6: Flag, IntFlag, auto
Creating an Enum

Enumerations are created using the `class` syntax, which makes them easy to read and write. An alternative creation method is described in Functional API. To define an enumeration, subclass `Enum` as follows:

```python
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
... 
```

**Note:** Enum member values

Member values can be anything: `int`, `str`, etc.. If the exact value is unimportant you may use `auto` instances and an appropriate value will be chosen for you. Care must be taken if you mix `auto` with other values.

**Note:** Nomenclature

- The class `Color` is an `enumeration` (or `enum`)
- The attributes `Color.RED`, `Color.GREEN`, etc., are `enumeration members` (or `enum members`) and are functionally constants.
- The enum members have `names` and `values` (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

**Note:** Even though we use the `class` syntax to create Enums, Enums are not normal Python classes. See How are Enums different? for more details.
Enumeration members have human readable string representations:

```python
>>> print(Color.RED)
Color.RED
```

...while their `repr` has more information:

```python
>>> print(repr(Color.RED))
<Color.RED: 1>
```

The `type` of an enumeration member is the enumeration it belongs to:

```python
>>> type(Color.RED)
<enum 'Color'>
```

Enumerations support iteration, in definition order:

```python
>>> class Shake(Enum):
    ...    VANILLA = 7
    ...    CHOCOLATE = 4
    ...    COOKIES = 9
    ...    MINT = 3
    ...

>>> for shake in Shake:
    ...    print(shake)
```
... 
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIE
Shake.MINT

Enumeration members are hashable, so they can be used in dictionaries and sets:

```python
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```
Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won't do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

If you want to access enum members by `name`, use item access:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If you have an enum member and need its name or value:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```
Duplicating enum members and values

Having two enum members with the same name is invalid:

```python
>>> class Shape(Enum):
    ...
    SQUARE  = 2
    ...
    SQUARE  = 3
    ...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

However, two enum members are allowed to have the same value. Given two members A and B with the same value (and A defined first), B is an alias to A. By-value lookup of the value of A and B will return A. By-name lookup of B will also return A:

```python
>>> class Shape(Enum):
    ...
    SQUARE  = 2
    ...
    DIAMOND = 1
    ...
    CIRCLE  = 3
    ...
    ALIAS_FOR_SQUARE  = 2
    ...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

**Note:** Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.
Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, the following decorator can be used to ensure each value is used only once in the enumeration:

```
@enum.unique
```

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__` gathering any aliases it finds; if any are found `ValueError` is raised with the details:

```python
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```
Using automatic values

If the exact value is unimportant you can use `auto`:

```python
>>> from enum import Enum, auto
>>> class Color(Enum):
...    RED = auto()
...    BLUE = auto()
...    GREEN = auto()
... >>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```python
>>> class AutoName(Enum):
...    def _generate_next_value_(name, start, count):
...        return name
... >>> class Ordinal(AutoName):
...    NORTH = auto()
...    SOUTH = auto()
...    EAST = auto()
...    WEST = auto()
... >>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>,
```

**Note:** The goal of the default `_generate_next_value_()` methods is to provide the next `int` in sequence with the last `int` provided, but the way it does this is an implementation detail and may change.
Iteration

Iterating over the members of an enum does not provide the aliases:

```python
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

The special attribute `__members__` is an ordered dictionary mapping names to members. It includes all names defined in the enumeration, including the aliases:

```python
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```python
>>> [name for name, member in Shape.__members__.items() if name == 'ALIAS_FOR_SQUARE']
['ALIAS_FOR_SQUARE']
```
Comparisons

Enumeration members are compared by identity:

```python
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see `IntEnum` below):

```python
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color'
```

Equality comparisons are defined though:

```python
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons against non-enumeration values will always compare not equal (again, `IntEnum` was explicitly designed to behave differently, see below):

```python
>>> Color.BLUE == 2
```
False
Allowed members and attributes of enumerations

The examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the Functional API), but not strictly enforced. In the vast majority of use-cases, one doesn’t care what the actual value of an enumeration is. But if the value is important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```python
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...     
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.
...     
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

Then:

```python
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
```
The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__`, `__add__`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__` and/or `__init__` then whatever value(s) were given to the enum member will be passed into those methods. See `Planet` for an example.
Restricted Enum subclassing

A new `Enum` class must have one base `Enum` class, up to one concrete data type, and as many `object`-based mixin classes as needed. The order of these base classes is:

```python
class EnumName([mix-in, ...], [data-type,] base-enum
    pass
```

Also, subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```python
>>> class MoreColor(Color):
    ...    PINK = 17
    ...
Traceback (most recent call last):
...  TypeError: Cannot extend enumerations
```

But this is allowed:

```python
>>> class Foo(Enum):
    ...    def some_behavior(self):
    ...        pass
    ...
>>> class Bar(Foo):
    ...    HAPPY = 1
    ...
    SAD = 2
    ...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior
between a group of enumerations. (See `OrderedEnum` for an example.)
Pickling

Enumerations can be pickled and unpickled:

```python
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO == loads(dumps(Fruit.TOMATO))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

**Note:** With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

It is possible to modify how Enum members are pickled/unpickled by defining `__reduce_ex__()` in the enumeration class.
Functional API

The `Enum` class is callable, providing the following functional API:

```python
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
```

The semantics of this API resemble `namedtuple`. The first argument of the call to `Enum` is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from `Enum` is returned. In other words, the above assignment to `Animal` is equivalent to:

```python
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
```
The reason for defaulting to 1 as the starting number and not 0 is that 0 is `False` in a boolean sense, but enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```python
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module='...
```

**Warning:** If `module` is not supplied, and `Enum` cannot determine what it is, the new `Enum` members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on `__qualname__` being set to the location where pickle will be able to find the class. For example, if the class was made available in class `SomeData` in the global scope:

```python
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='...
```

The complete signature is:

```python
Enum(value='NewEnumName', names=<..., *, module='...
```
<table>
<thead>
<tr>
<th><strong>value:</strong></th>
<th>What the new Enum class will record as its name.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>names:</strong></td>
<td>The Enum members. This can be a whitespace or comma-separated string (values will start at 1 unless otherwise specified):</td>
</tr>
<tr>
<td></td>
<td>'RED_GREEN_BLUE'</td>
</tr>
<tr>
<td></td>
<td>or an iterator of names:</td>
</tr>
<tr>
<td></td>
<td>[ 'RED', 'GREEN', 'BLUE' ]</td>
</tr>
<tr>
<td></td>
<td>or an iterator of (name, value) pairs:</td>
</tr>
<tr>
<td></td>
<td>[ ('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 10) ]</td>
</tr>
<tr>
<td></td>
<td>or a mapping:</td>
</tr>
<tr>
<td></td>
<td>{ 'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 12 }</td>
</tr>
<tr>
<td><strong>module:</strong></td>
<td>name of module where new Enum class can be found.</td>
</tr>
<tr>
<td><strong>qualname:</strong></td>
<td>where in module new Enum class can be found.</td>
</tr>
<tr>
<td><strong>type:</strong></td>
<td>type to mix in to new Enum class.</td>
</tr>
<tr>
<td><strong>start:</strong></td>
<td>number to start counting at if only names are passed in.</td>
</tr>
</tbody>
</table>

*Changed in version 3.5:* The *start* parameter was added.
Derived Enumerations

IntEnum

The first variation of Enum that is provided is also a subclass of int. Members of an IntEnum can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```python
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...    CIRCLE = 1
...    SQUARE = 2
...   
>>> class Request(IntEnum):
...    POST = 1
...    GET = 2
...   
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, they still can’t be compared to standard Enum enumerations:

```python
>>> class Shape(IntEnum):
...    CIRCLE = 1
...    SQUARE = 2
...   
>>> class Color(Enum):
...    RED = 1
...    GREEN = 2
```
\[\text{Shape}.\text{CIRCLE} == \text{Color}.\text{RED}\]

IntEnum values behave like integers in other ways you’d expect:

\[\text{int}((\text{Shape}.\text{CIRCLE})\]
\n1
\n\[\text{['a', 'b', 'c'][\text{Shape}.\text{CIRCLE}]}\]
\n'b'
\n\[\text{[i for i in range(\text{Shape}.\text{SQUARE})]}\]
\n[0, 1]

IntFlag

The next variation of Enum provided, IntFlag, is also based on int. The difference being IntFlag members can be combined using the bitwise operators (\&, |, ^, ~) and the result is still an IntFlag member. However, as the name implies, IntFlag members also subclass int and can be used wherever an int is used. Any operation on an IntFlag member besides the bit-wise operations will lose the IntFlag membership.

New in version 3.6.

Sample IntFlag class:

\[\text{from enum import IntFlag}\]
\n\[\text{class Perm(IntFlag):}\]
\n\[\text{... R = 4}\]
\n\[\text{... W = 2}\]
\n\[\text{... X = 1}\]
\n\[\text{...}\]
\n\[\text{Perm.R | Perm.W}\]
It is also possible to name the combinations:

```python
class Perm(IntFlag):
    ...  R = 4
    ...  W = 2
    ...  X = 1
    ...  RWX = 7

Perm.RWX
<Perm.RWX: 7>
~Perm.RWX
<Perm.-8: -8>
```

Another important difference between `IntFlag` and `Enum` is that if no flags are set (the value is 0), its boolean evaluation is `False`:

```python
Perm.R & Perm.X
<Perm.0: 0>
bool(Perm.R & Perm.X)
False
```

Because `IntFlag` members are also subclasses of `int` they can be combined with them:

```python
Perm.X | 8
<Perm.8|X: 9>
```

Flag
The last variation is `Flag`. Like `IntFlag`, `Flag` members can be combined using the bitwise operators (&, |, ^, ~). Unlike `IntFlag`, they cannot be combined with, nor compared against, any other `Flag` enumeration, nor `int`. While it is possible to specify the values directly it is recommended to use `auto` as the value and let `Flag` select an appropriate value.

*New in version 3.6.*

Like `IntFlag`, if a combination of `Flag` members results in no flags being set, the boolean evaluation is `False`:

```python
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, ...), while combinations of flags won’t:

```python
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```
Giving a name to the “no flags set” condition does not change its boolean value:

```python
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

**Note:** For the majority of new code, `Enum` and `Flag` are strongly recommended, since `IntEnum` and `IntFlag` break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). `IntEnum` and `IntFlag` should be used only in cases where `Enum` and `Flag` will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

**Others**

While `IntEnum` is part of the `enum` module, it would be very simple to implement independently:

```python
class IntEnum(int, Enum):
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a `StrEnum` that mixes in `str` instead of `int`. 
Some rules:

1. When subclassing `Enum`, mix-in types must appear before `Enum` itself in the sequence of bases, as in the `IntEnum` example above.
2. While `Enum` can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. `int` above. This restriction does not apply to mix-ins which only add methods and don’t specify another data type such as `int` or `str`.
3. When another data type is mixed in, the value attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
4. %-style formatting: `%s` and `%r` call the `Enum` class’s `__str__()` and `__repr__()` respectively; other codes (such as `%i` or `%h` for `IntEnum`) treat the enum member as its mixed-in type.
5. Formatted string literals, `str.format()`, and `format()` will use the mixed-in type’s `__format__()`. If the `Enum` class’s `str()` or `repr()` is desired, use the `!s` or `!r` format codes.
Interesting examples

While `Enum`, `IntEnum`, `IntFlag`, and `Flag` are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one’s own.

Omitting values

In many use-cases one doesn’t care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of `auto` for the value
- use instances of `object` as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Whichever method you choose, you should provide a `repr()` that also hides the (unimportant) value:

```python
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...```
Using **auto**

Using `auto` would look like:

```python
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Using **object**

Using `object` would look like:

```python
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```python
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
```

Using a custom `__new__()`

Using an auto-numbering `__new__()` would look like:

```python
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj

>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()

>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

**Note:** The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum’s `__new__()` which is used after class creation for lookup of existing members.

**OrderedEnum**

An ordered enumeration that is not based on `IntEnum` and so maintains the normal `Enum` invariants (such as not being comparable to other enumerations):
>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented

>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1

>>> Grade.C < Grade.A
True

DuplicateFreeEnum

Raises an error if a duplicate member name is found instead of creating an alias:

>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         pass
...    cls = self.__class__
...    if any(self.value == e.value for e in cls):
...        a = self.name
...        e = cls(self.value).name
...        raise ValueError(
...            "aliases not allowed in DuplicateFreeEnum:
...            % (a, e))
...

>>> class Color(DuplicateFreeEnum):
...    RED = 1
...    GREEN = 2
...    BLUE = 3
...    GRENE = 2
...

Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:

Note: This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the unique() decorator can be used instead.

Planet

If __new__() or __init__() is defined the value of the enum member will be passed to those methods:

>>> class Planet(Enum):
...    MERCURY = (3.303e+23, 2.4397e6)
...    VENUS = (4.869e+24, 6.0518e6)
...    EARTH = (5.976e+24, 6.37814e6)
...    MARS = (6.421e+23, 3.3972e6)
...    JUPITER = (1.9e+27, 7.1492e7)
... SATURN = (5.688e+26, 6.0268e7)
... URANUS = (8.686e+25, 2.5559e7)
... NEPTUNE = (1.024e+26, 2.4746e7)
... def __init__(self, mass, radius):
...     self.mass = mass  # in kilograms
...     self.radius = radius  # in meters
... @property
... def surface_gravity(self):
...     # universal gravitational constant (m^3 kg^-1 s^-2)
...     G = 6.67300E-11
...     return G * self.mass / (self.radius * self.radius)

>>> Planet.EARTH.value
(5.976e+24, 6378140.0)

>>> Planet.EARTH.surface_gravity
9.802652743337129

TimePeriod

An example to show the _ignore_ attribute in use:

>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i

>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]

>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.timedelta(days=366)>]
How are Enums different?

 Enums have a custom metaclass that affects many aspects of both derived Enum classes and their instances (members).

Enum Classes

The EnumMeta metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an `Enum` class that fail on a typical class, such as `list(Color)` or `some_enum_var in Color`. EnumMeta is responsible for ensuring that various other methods on the final `Enum` class are correct (such as `__new__()`, `__getnewargs__()`, `__str__()` and `__repr__()`).

Enum Members (aka instances)

The most interesting thing about Enum members is that they are singletons. EnumMeta creates them all while it is creating the `Enum` class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Finer Points

Supported `__dunder__` names

`__members__` is an OrderedDict of `member_name:member` items. It is only available on the class.
__new__(), if specified, must create and return the enum members; it is also a very good idea to set the member's _value_ appropriately. Once all the members are created it is no longer used.

Supported _sunder_ names

- _name_ – name of the member
- _value_ – value of the member; can be set / modified in __new__
- _missing_ – a lookup function used when a value is not found; may be overridden
- _ignore_ – a list of names, either as a list() or a str(), that will not be transformed into members, and will be removed from the final class
- _order_ – used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- _generate_next_value_ – used by the Functional API and by auto to get an appropriate value for an enum member; may be overridden

New in version 3.6: _missing_, _order_, _generate_next_value_

New in version 3.7: _ignore_

To help keep Python 2 / Python 3 code in sync an _order_ attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```python
class Color(Enum):
    ...
    _order_ = 'RED GREEN BLUE'
    ...
    RED = 1
    ...
    BLUE = 3
```
... GREEN = 2
...

Traceback (most recent call last):
...
TypeError: member order does not match _order_

**Note:** In Python 2 code the _order_ attribute is necessary as definition order is lost before it can be recorded.

## Enum member type

Enum members are instances of their `Enum` class, and are normally accessed as `EnumClass.member`. Under certain circumstances they can also be accessed as `EnumClass.member.member`, but you should never do this as that lookup may fail or, worse, return something besides the `Enum` member you are looking for (this is another good reason to use all-uppercase names for members):

```python
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

*Changed in version 3.5.*

## Boolean value of Enum classes and members

Enum members that are mixed with non-`Enum` types (such as `int`,
str, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as True. To make your own Enum's boolean evaluation depend on the member's value add the following to your class:

```python
def __bool__(self):
    return bool(self.value)
```

Enum classes always evaluate as True.

Enum classes with methods

If you give your Enum subclass extra methods, like the Planet class above, those methods will show up in a dir() of the member, but not of the class:

```python
>>> dir(People)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__class__', '__doc__', '__members__', '__module__']
>>> dir(People.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combining members of Flag

If a combination of Flag members is not named, the repr() will include all named flags and all named combinations of flags that are in the value:

```python
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
```
... CYAN = GREEN | BLUE
...

>>> Color(3) # named combination
<Color.YELLOW: 3>

>>> Color(7) # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>

© Copyright 2001-2019, Python Software Foundation.  
The Python Software Foundation is a non-profit corporation. Please donate.  
Last updated on Mar 25, 2019. Found a bug?  
Created using Sphinx 1.8.5.
Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. The decimal module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

The following modules are documented in this chapter:

- numbers — Numeric abstract base classes
  - The numeric tower
  - Notes for type implementors
    - Adding More Numeric ABCs
    - Implementing the arithmetic operations
- math — Mathematical functions
  - Number-theoretic and representation functions
  - Power and logarithmic functions
  - Trigonometric functions
  - Angular conversion
  - Hyperbolic functions
  - Special functions
  - Constants
- cmath — Mathematical functions for complex numbers
  - Conversions to and from polar coordinates
  - Power and logarithmic functions
  - Trigonometric functions
  - Hyperbolic functions
  - Classification functions
- Constants
  - **decimal** — Decimal fixed point and floating point arithmetic
    - Quick-start Tutorial
    - Decimal objects
      - Logical operands
    - Context objects
    - Constants
    - Rounding modes
    - Signals
    - Floating Point Notes
      - Mitigating round-off error with increased precision
      - Special values
    - Working with threads
    - Recipes
    - Decimal FAQ
- **fractions** — Rational numbers
- **random** — Generate pseudo-random numbers
  - Bookkeeping functions
  - Functions for integers
  - Functions for sequences
  - Real-valued distributions
  - Alternative Generator
  - Notes on Reproducibility
  - Examples and Recipes
- **statistics** — Mathematical statistics functions
  - Averages and measures of central location
  - Measures of spread
  - Function details
  - Exceptions
numbers — Numeric abstract base classes

Source code: Lib/numbers.py

The numbers module (PEP 3141) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module can be instantiated.

class numbers.Number
    The root of the numeric hierarchy. If you just want to check if an argument \( x \) is a number, without caring what kind, use \( \text{isinstance}(x, \text{Number}) \).
The numeric tower

class numbers.Complex
Subclasses of this type describe complex numbers and include the operations that work on the built-in complex type. These are: conversions to complex and bool, real, imag, +, -, *, /, abs(), conjugate(), ==, and !=. All except - and != are abstract.

real
Abstract. Retrieves the real component of this number.

imag
Abstract. Retrieves the imaginary component of this number.

abstractmethod conjugate()
Abstract. Returns the complex conjugate. For example, (1+3j).conjugate() == (1-3j).

class numbers.Real
To Complex, Real adds the operations that work on real numbers.

In short, those are: a conversion to float, math.trunc(), round(), math.floor(), math.ceil(), divmod(), //, %, <, <=, >, and >=.

Real also provides defaults for complex(), real, imag, and conjugate().

class numbers.Rational
Subtypes Real and adds numerator and denominator
properties, which should be in lowest terms. With these, it provides a default for \texttt{float()}.

\textbf{numerator}

Abstract.

\textbf{denominator}

Abstract.

\textit{class} numbers. \textbf{Integral}

Subtypes \texttt{Rational} and adds a conversion to \texttt{int}. Provides defaults for \texttt{float()}, \texttt{numerator}, and \texttt{denominator}. Adds abstract methods for ** and bit-string operations: \texttt{<<, >>, &}, \texttt{\&\&}, \texttt{|}, \texttt{\|}, \texttt{\~{}.}
Notes for type implementors

Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, `fractions.Fraction` implements `hash()` as follows:

```python
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add `MyFoo` between `Complex` and `Real` with:

```python
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode
operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of `Integral`, this means that `__add__()` and `__radd__()` should be defined as:

```python
class MyIntegral(Integral):
    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding Stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of `Complex`. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as “boilerplate”. `a` will be an instance of `A`, which is a subtype of `Complex` (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:
1. If A defines an `__add__()` which accepts b, all is well.
2. If A falls back to the boilerplate code, and it were to return a value from `__add__()`, we’d miss the possibility that B defines a more intelligent `__radd__()`, so the boilerplate should return `NotImplemented` from `__add__()`. (Or A may not implement `__add__()` at all.)
3. Then B’s `__radd__()` gets a chance. If it accepts a, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If B <: A, Python tries B.__radd__ before A.__add__. This is ok, because it was implemented with knowledge of A, so it can handle those instances before delegating to `Complex`.

If A <: Complex and B <: Real without sharing any other knowledge, then the appropriate shared operation is the one involving the built in `complex`, and both `__radd__()` s land there, so a+b == b+a.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, `fractions.Fraction` uses:

```python
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
```
else:
    return NotImplemented

forward.__name__ = '__' + fallback_operator.__name__
forward.__doc__ = monomorphic_operator.__doc__

def reverse(b, a):
    if isinstance(a, Rational):
        # Includes ints.
        return monomorphic_operator(a, b)
    elif isinstance(a, numbers.Real):
        return fallback_operator(float(a), float(b))
    elif isinstance(a, numbers.Complex):
        return fallback_operator(complex(a), complex(b))
    else:
        return NotImplemented

reverse.__name__ = '__r' + fallback_operator.__name__
reverse.__doc__ = monomorphic_operator.__doc__

return forward, reverse

def __add__(a, b):
    """a + b""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(__add__, operator)

# ...
**math — Mathematical functions**

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.
Number-theoretic and representation functions

math.ceil(x)

Return the ceiling of x, the smallest integer greater than or equal to x. If x is not a float, delegates to x.__ceil__(), which should return an Integral value.

math.copysign(x, y)

Return a float with the magnitude (absolute value) of x but the sign of y. On platforms that support signed zeros, copysign(1.0, -0.0) returns -1.0.

math.fabs(x)

Return the absolute value of x.

math.factorial(x)

Return x factorial. Raises ValueError if x is not integral or is negative.

math.floor(x)

Return the floor of x, the largest integer less than or equal to x. If x is not a float, delegates to x.__floor__(), which should return an Integral value.

math.fmod(x, y)

Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result. The intent of the C standard is that fmod(x, y) be exactly (mathematically; to infinite precision) equal to x - n*y for some
integer $n$ such that the result has the same sign as $x$ and magnitude less than $\text{abs}(y)$. Python’s $x \mod y$ returns a result with the sign of $y$ instead, and may not be exactly computable for float arguments. For example, $\text{fmod}(-1e-100, 1e100)$ is $-1e-100$, but the result of Python’s $-1e-100 \mod 1e100$ is $1e100-1e-100$, which cannot be represented exactly as a float, and rounds to the surprising $1e100$. For this reason, function $\text{fmod()}$ is generally preferred when working with floats, while Python’s $x \mod y$ is preferred when working with integers.

**math.\text{frexp}(x)$**

Return the mantissa and exponent of $x$ as the pair $(m, e)$. $m$ is a float and $e$ is an integer such that $x == m \times 2^{e}$ exactly. If $x$ is zero, returns $(0.0, 0)$, otherwise $0.5 \leq \text{abs}(m) < 1$. This is used to “pick apart” the internal representation of a float in a portable way.

**math.\text{fsum}(\text{iterable})**

Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm’s accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.
For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

**math.\texttt{gcd}(a, b)**

Return the greatest common divisor of the integers \( a \) and \( b \). If either \( a \) or \( b \) is nonzero, then the value of \( \text{gcd}(a, b) \) is the largest positive integer that divides both \( a \) and \( b \). \( \text{gcd}(0, 0) \) returns 0.

*New in version 3.5.*

**math.\texttt{isclose}(a, b, *, rel\_tol=1e-09, abs\_tol=0.0)**

Return True if the values \( a \) and \( b \) are close to each other and False otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

\( \text{rel\_tol} \) is the relative tolerance – it is the maximum allowed difference between \( a \) and \( b \), relative to the larger absolute value of \( a \) or \( b \). For example, to set a tolerance of 5%, pass \( \text{rel\_tol}=0.05 \). The default tolerance is 1e-09, which assures that the two values are the same within about 9 decimal digits. \( \text{rel\_tol} \) must be greater than zero.

\( \text{abs\_tol} \) is the minimum absolute tolerance – useful for comparisons near zero. \( \text{abs\_tol} \) must be at least zero.

If no errors occur, the result will be: \( \text{abs}(a-b) \leq \max(\text{rel\_tol} \* \max(\text{abs}(a), \text{abs}(b)), \text{abs\_tol}) \).

The IEEE 754 special values of NaN, inf, and -inf will be handled according to IEEE rules. Specifically, NaN is not
considered close to any other value, including NaN. inf and -inf are only considered close to themselves.

*New in version 3.5.*

**See also:** PEP 485 – A function for testing approximate equality

```python
math.isfinite(x)
```

Return `True` if `x` is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` is considered finite.)

*New in version 3.2.*

```python
math.isinf(x)
```

Return `True` if `x` is a positive or negative infinity, and `False` otherwise.

```python
math.isnan(x)
```

Return `True` if `x` is a NaN (not a number), and `False` otherwise.

```python
math.ldexp(x, i)
```

Return `x * (2**i)`. This is essentially the inverse of function `frexp()`.

```python
math.modf(x)
```

Return the fractional and integer parts of `x`. Both results carry the sign of `x` and are floats.

```python
math.remainder(x, y)
```

Return the IEEE 754-style remainder of `x` with respect to `y`. For finite `x` and finite nonzero `y`, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If
\[ \frac{x}{y} \] is exactly halfway between two consecutive integers, the nearest even integer is used for \( n \). The remainder \( r = \text{remainder}(x, y) \) thus always satisfies \( |r| \leq 0.5 \times |y| \).

Special cases follow IEEE 754: in particular, \( \text{remainder}(x, \text{math.inf}) \) is \( x \) for any finite \( x \), and \( \text{remainder}(x, 0) \) and \( \text{remainder}(\text{math.inf}, x) \) raise \text{ValueError} for any non-NaN \( x \). If the result of the remainder operation is zero, that zero will have the same sign as \( x \).

On platforms using IEEE 754 binary floating-point, the result of this operation is always exactly representable: no rounding error is introduced.

*New in version 3.7.*

**math.\texttt{trunc}(x)**

Return the Real value \( x \) truncated to an Integral (usually an integer). Delegates to \( x.__\text{trunc__}() \).

Note that \text{frexp()} and \text{modf()} have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an ‘output parameter’ (there is no such thing in Python).

For the \text{ceil()}, \text{floor()}, and \text{modf()} functions, note that all floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float \( x \) with \( \text{abs}(x) \geq 2^{*52} \) necessarily has no fractional bits.
Power and logarithmic functions

**math.exp(x)**

Return \(e\) raised to the power \(x\), where \(e = 2.718281\ldots\) is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

**math.expm1(x)**

Return \(e\) raised to the power \(x\), minus 1. Here \(e\) is the base of natural logarithms. For small floats \(x\), the subtraction in `exp(x) - 1` can result in a significant loss of precision; the `expm1()` function provides a way to compute this quantity to full precision:

```python
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

*New in version 3.2.*

**math.log(x[, base])**

With one argument, return the natural logarithm of \(x\) (to base \(e\)).

With two arguments, return the logarithm of \(x\) to the given `base`, calculated as `log(x)/log(base)`.

**math.log1p(x)**

Return the natural logarithm of \(1+x\) (base \(e\)). The result is calculated in a way which is accurate for \(x\) near zero.

**math.log2(x)**
Return the base-2 logarithm of \( x \). This is usually more accurate than \( \log(x, 2) \).

*New in version 3.3.*

**See also:** `int.bit_length()` returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

**math.\texttt{log10}(x)**

Return the base-10 logarithm of \( x \). This is usually more accurate than \( \log(x, 10) \).

**math.\texttt{pow}(x, y)**

Return \( x \) raised to the power \( y \). Exceptional cases follow Annex ‘F’ of the C99 standard as far as possible. In particular, \( \texttt{pow(1.0, x)} \) and \( \texttt{pow(x, 0.0)} \) always return 1.0, even when \( x \) is a zero or a NaN. If both \( x \) and \( y \) are finite, \( x \) is negative, and \( y \) is not an integer then \( \texttt{pow(x, y)} \) is undefined, and raises `ValueError`.

Unlike the built-in `**` operator, `math.pow()` converts both its arguments to type `float`. Use `**` or the built-in `pow()` function for computing exact integer powers.

**math.\texttt{sqrt}(x)**

Return the square root of \( x \).
Trigonometric functions

math.\texttt{acos}(x)
  Return the arc cosine of $x$, in radians.

math.\texttt{asin}(x)
  Return the arc sine of $x$, in radians.

math.\texttt{atan}(x)
  Return the arc tangent of $x$, in radians.

math.\texttt{atan2}(y, x)
  Return $\text{atan}(y / x)$, in radians. The result is between $-\pi$ and $\pi$. The vector in the plane from the origin to point $(x, y)$ makes this angle with the positive X axis. The point of \texttt{atan2()} is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, $\text{atan}(1)$ and $\text{atan2}(1, 1)$ are both $\pi/4$, but $\text{atan2}(-1, -1)$ is $-3\pi/4$.

math.\texttt{cos}(x)
  Return the cosine of $x$ radians.

math.\texttt{hypot}(x, y)
  Return the Euclidean norm, $\sqrt{x^2 + y^2}$. This is the length of the vector from the origin to point $(x, y)$.

math.\texttt{sin}(x)
  Return the sine of $x$ radians.

math.\texttt{tan}(x)
  Return the tangent of $x$ radians.
Angular conversion

\texttt{math.degrees(x)}

Convert angle $x$ from radians to degrees.

\texttt{math.radians(x)}

Convert angle $x$ from degrees to radians.
Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

\texttt{math.acosh(x)}

Return the inverse hyperbolic cosine of \( x \).

\texttt{math.asinh(x)}

Return the inverse hyperbolic sine of \( x \).

\texttt{math.atanh(x)}

Return the inverse hyperbolic tangent of \( x \).

\texttt{math.cosh(x)}

Return the hyperbolic cosine of \( x \).

\texttt{math.sinh(x)}

Return the hyperbolic sine of \( x \).

\texttt{math.tanh(x)}

Return the hyperbolic tangent of \( x \).
**Special functions**

`math.erf(x)`  
Return the error function at x.

The `erf()` function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```python
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

*New in version 3.2.*

`math.erfc(x)`  
Return the complementary error function at x. The complementary error function is defined as \( 1.0 - erf(x) \). It is used for large values of x where a subtraction from one would cause a loss of significance.

*New in version 3.2.*

`math.gamma(x)`  
Return the Gamma function at x.

*New in version 3.2.*

`math.lgamma(x)`  
Return the natural logarithm of the absolute value of the Gamma function at x.

*New in version 3.2.*
Constants

math.pi

The mathematical constant $\pi = 3.141592\ldots$, to available precision.

math.e

The mathematical constant $e = 2.718281\ldots$, to available precision.

math.tau

The mathematical constant $\tau = 6.283185\ldots$, to available precision. Tau is a circle constant equal to $2\pi$, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

*New in version 3.6.*

math.inf

A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf').

*New in version 3.5.*

math.nan

A floating-point “not a number” (NaN) value. Equivalent to the output of float('nan').

*New in version 3.5.*

**CPython implementation detail:** The math module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard
where appropriate. The current implementation will raise `ValueError` for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and `OverflowError` for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

**See also:**

**Module `cmath`**

Complex number versions of many of these functions.
cmath — Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a \_\_complex\_\_() or a \_\_float\_\_() method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

**Note:** On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on both sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros the continuity is as specified below.
Conversions to and from polar coordinates

A Python complex number $z$ is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* $z$.real and its *imaginary part* $z$.imag. In other words:

$$z = z$.real + z$.imag * 1j$$

_Polar coordinates_ give an alternative way to represent a complex number. In polar coordinates, a complex number $z$ is defined by the modulus $r$ and the phase angle $\phi$. The modulus $r$ is the distance from $z$ to the origin, while the phase $\phi$ is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to $z$.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

**cmath.phase(x)**

Return the phase of $x$ (also known as the *argument* of $x$), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range $[-\pi, \pi]$, and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of $x$.imag, even when $x$.imag is zero:

```python
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```
Note: The modulus (absolute value) of a complex number $x$ can be computed using the built-in \texttt{abs()} function. There is no separate \texttt{cmath} module function for this operation.

\texttt{cmath.polar}(x)

Return the representation of $x$ in polar coordinates. Returns a pair $(r, \phi)$ where $r$ is the modulus of $x$ and $\phi$ is the phase of $x$. \texttt{polar}(x) is equivalent to \texttt{(abs(x), phase(x))}.

\texttt{cmath.rect}(r, \phi)

Return the complex number $x$ with polar coordinates $r$ and $\phi$. Equivalent to $r \times \texttt{(math.cos(\phi) + math.sin(\phi)*1j)}$. 
Power and logarithmic functions

cmath.\texttt{exp}(x)

Return $e$ raised to the power $x$, where $e$ is the base of natural logarithms.

cmath.\texttt{log}(x[,\ base])

Returns the logarithm of $x$ to the given \texttt{base}. If the \texttt{base} is not specified, returns the natural logarithm of $x$. There is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.

cmath.\texttt{log10}(x)

Return the base-10 logarithm of $x$. This has the same branch cut as \texttt{log()}.

cmath.\texttt{sqrt}(x)

Return the square root of $x$. This has the same branch cut as \texttt{log()}. 
Trigonometric functions

cmath.\texttt{acos}(x)

Return the arc cosine of \( x \). There are two branch cuts: One extends right from 1 along the real axis to \( \infty \), continuous from below. The other extends left from -1 along the real axis to -\( \infty \), continuous from above.

cmath.\texttt{asin}(x)

Return the arc sine of \( x \). This has the same branch cuts as \texttt{acos()}.

cmath.\texttt{atan}(x)

Return the arc tangent of \( x \). There are two branch cuts: One extends from \( 1j \) along the imaginary axis to \( \infty j \), continuous from the right. The other extends from \( -1j \) along the imaginary axis to -\( \infty j \), continuous from the left.

cmath.\texttt{cos}(x)

Return the cosine of \( x \).

cmath.\texttt{sin}(x)

Return the sine of \( x \).

cmath.\texttt{tan}(x)

Return the tangent of \( x \).
Hyperbolic functions

cmath.\textbf{acosh}(x)

Return the inverse hyperbolic cosine of \(x\). There is one branch cut, extending left from 1 along the real axis to \(-\infty\), continuous from above.

cmath.\textbf{asinh}(x)

Return the inverse hyperbolic sine of \(x\). There are two branch cuts: One extends from \(1j\) along the imaginary axis to \(\infty j\), continuous from the right. The other extends from \(-1j\) along the imaginary axis to \(-\infty j\), continuous from the left.

cmath.\textbf{atanh}(x)

Return the inverse hyperbolic tangent of \(x\). There are two branch cuts: One extends from \(1\) along the real axis to \(\infty\), continuous from below. The other extends from \(-1\) along the real axis to \(-\infty\), continuous from above.

cmath.\textbf{cosh}(x)

Return the hyperbolic cosine of \(x\).

cmath.\textbf{sinh}(x)

Return the hyperbolic sine of \(x\).

cmath.\textbf{tanh}(x)

Return the hyperbolic tangent of \(x\).
Classification functions

cmath.\texttt{isfinite}(x)

Return \texttt{True} if both the real and imaginary parts of \(x\) are finite, and \texttt{False} otherwise.

\textit{New in version 3.2.}

cmath.\texttt{isinf}(x)

Return \texttt{True} if either the real or the imaginary part of \(x\) is an infinity, and \texttt{False} otherwise.

cmath.\texttt{isnan}(x)

Return \texttt{True} if either the real or the imaginary part of \(x\) is a NaN, and \texttt{False} otherwise.

cmath.\texttt{isclose}(a, b, *, rel\_tol=1e-09, abs\_tol=0.0)

Return \texttt{True} if the values \(a\) and \(b\) are close to each other and \texttt{False} otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

\textit{rel\_tol} is the relative tolerance – it is the maximum allowed difference between \(a\) and \(b\), relative to the larger absolute value of \(a\) or \(b\). For example, to set a tolerance of 5%, pass \textit{rel\_tol}=0.05. The default tolerance is 1e-09, which assures that the two values are the same within about 9 decimal digits. \textit{rel\_tol} must be greater than zero.

\textit{abs\_tol} is the minimum absolute tolerance – useful for comparisons near zero. \textit{abs\_tol} must be at least zero.
If no errors occur, the result will be: \( \text{abs}(a-b) \leq \max(\text{rel} \_ \text{tol} \times \max(\text{abs}(a), \text{abs}(b)), \text{abs} \_ \text{tol}) \).

The IEEE 754 special values of \( \text{NaN} \), \( \text{inf} \), and \( -\text{inf} \) will be handled according to IEEE rules. Specifically, \( \text{NaN} \) is not considered close to any other value, including \( \text{NaN} \). \( \text{inf} \) and \( -\text{inf} \) are only considered close to themselves.

*New in version 3.5.*

**See also:** PEP 485 – A function for testing approximate equality
Constants

cmath.\texttt{pi}

The mathematical constant $\pi$, as a float.

cmath.\texttt{e}

The mathematical constant $e$, as a float.

cmath.\texttt{tau}

The mathematical constant $\tau$, as a float.

\textit{New in version 3.6.}

cmath.\texttt{inf}

Floating-point positive infinity. Equivalent to \texttt{float('inf')}.

\textit{New in version 3.6.}

cmath.\texttt{infj}

Complex number with zero real part and positive infinity imaginary part. Equivalent to \texttt{complex(0.0, float('inf'))}.

\textit{New in version 3.6.}

cmath.\texttt{nan}

A floating-point “not a number” (NaN) value. Equivalent to \texttt{float('nan')}.

\textit{New in version 3.6.}

cmath.\texttt{nanj}

Complex number with zero real part and NaN imaginary part. Equivalent to \texttt{complex(0.0, float('nan'))}.
New in version 3.6.

Note that the selection of functions is similar, but not identical, to that in module `math`. The reason for having two modules is that some users aren’t interested in complex numbers, and perhaps don’t even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in `cmath` always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

**decimal** — Decimal fixed point and floating point arithmetic

**Source code:** Lib/decimal.py

The `decimal` module provides support for fast correctly-rounded decimal floating point arithmetic. It offers several advantages over the `float` datatype:

- Decimal “is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect 1.1 + 2.2 to display as 3.3000000000000003 as it does with binary floating point.

- The exactness carries over into arithmetic. In decimal floating point, 0.1 + 0.1 + 0.1 - 0.3 is exactly equal to zero. In binary floating point, the result is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

- The decimal module incorporates a notion of significant places so
that $1.30 + 1.20$ is $2.50$. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, $1.3 \times 1.2$ gives $1.56$ while $1.30 \times 1.20$ gives $1.5600$.

- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```python
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

- The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.
A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, and trap enablers which determine whether signals are treated as exceptions. Rounding options include ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, ROUND_UP, and ROUND_05UP.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: Clamped, InvalidOperation, DivisionByZero, Inexact, Rounded, Subnormal, Overflow, Underflow and FloatOperation.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also:

Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```python
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero, InvalidOperation])

>>> getcontext().prec = 7               # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as NaN which stands for “Not a number”, positive and negative Infinity, and -0:

```python
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446748141679699347203125')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
```
If the `FloatOperation` signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```python
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

*New in version 3.3.*

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```python
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
```

```python
Decimal('5.860000')
```
Decimal('5.85987')

>>> getcontext().rounding = ROUND_UP

>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')

If the internal limits of the C version are exceeded, constructing a decimal raises `InvalidOperation`:

```python
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

*Changed in version 3.3.*

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```python
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'))
>>> max(data)
Decimal('9.25')

>>> min(data)
Decimal('0.03')

>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'), Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]

>>> sum(data)
Decimal('19.29')

>>> a, b, c = data[:3]

>>> str(a)
'1.34'

>>> float(a)
1.34

>>> round(a, 1)
Decimal('1.3')
```
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')

And some mathematical functions are also available to Decimal:

>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488017991455')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')

The quantize() method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_HALF_UP)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_HALF_UP)
Decimal('8')

As shown above, the getcontext() function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.
For more advanced work, it may be useful to create alternate contexts using the `Context()` constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `decimal` module provides two ready to use standard contexts, `BasicContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled:

```python
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[])  # Context with extended precision and rounding mode
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.
The `flags` entry shows that the rational approximation to Pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` field of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to `Decimal` with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.
Decimal objects

```python
class decimal.Decimal(value="0", context=None)

Construct a new Decimal object based from value.
```

*value* can be an integer, string, tuple, `float`, or another Decimal object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

```plaintext
sign ::= '+' | '-'
digit ::= '0' | '1' | '2' | '3' | '4' |
indicator ::= 'e' | 'E'
digits ::= digit [digit]...
decimal-part ::= digits '.' [digits] | ['.']
exponent-part ::= indicator [sign] digits
infinity ::= 'Infinity' | 'Inf'
nan ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value ::= decimal-part [exponent-part]
numeric-string ::= [sign] numeric-value | [sign]
```

Other Unicode decimal digits are also permitted where *digit* appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `\uff10` through `\uff19`.

If *value* is a `tuple`, it should have three components, a sign (0 for positive or 1 for negative), a `tuple` of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`. 
If \textit{value} is a \texttt{float}, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, \texttt{Decimal(float('1.1'))} converts to \texttt{Decimal('1.100000000000000088817841970012523233890')}

The \textit{context} precision does not affect how many digits are stored. That is determined exclusively by the number of digits in \textit{value}. For example, \texttt{Decimal('3.00000')} records all five zeros even if the context precision is only three.

The purpose of the \textit{context} argument is determining what to do if \textit{value} is a malformed string. If the context traps \texttt{InvalidOperation}, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, \texttt{Decimal} objects are immutable.

\textit{Changed in version 3.2}: The argument to the constructor is now permitted to be a \texttt{float} instance.

\textit{Changed in version 3.3}: \texttt{float} arguments raise an exception if the \texttt{FloatOperation} trap is set. By default the trap is off.

\textit{Changed in version 3.6}: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating point objects share many properties with the other built-in numeric types such as \texttt{float} and \texttt{int}. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as \texttt{float} or \texttt{int}).
There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator \% is applied to Decimal objects, the sign of the result is the sign of the *dividend* rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator \// behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity $x == (x \div y) * y + x \% y$:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The \% and \// operators implement the *remainder* and *divide-integer* operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of `fractions.Fraction` in arithmetic operations: an attempt to add a `Decimal` to a `float`, for example, will raise a `TypeError`. However, it is possible to use Python's comparison operators to compare a `Decimal` instance $x$ with another number $y$. This avoids confusing results when doing equality comparisons between numbers of different types.

*Changed in version 3.2:* Mixed-type comparisons between
Decimal instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()

Return a pair \((n, d)\) of integers that represent the given `Decimal` instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

The conversion is exact. Raise `OverflowError` on infinities and `ValueError` on NaNs.

*New in version 3.6.*

as_tuple()

Return a `named tuple` representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a `Decimal` instance is always canonical, so this
operation returns its argument unchanged.

**compare**(other, context=None)

Compare the values of two Decimal instances. `compare()` returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a or b is a NaN</td>
<td>Decimal('NaN')</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>Decimal('-1')</td>
</tr>
<tr>
<td>a == b</td>
<td>Decimal('0')</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>Decimal('1')</td>
</tr>
</tbody>
</table>

**compare_signal**(other, context=None)

This operation is identical to the `compare()` method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

**compare_total**(other, context=None)

Compare two operands using their abstract representation rather than their numerical value. Similar to the `compare()` method, but the result gives a total ordering on Decimal instances. Two Decimal instances with the same numeric value but different representations compare unequal in this ordering:

```python
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is Decimal('0') if both operands have the same representation, Decimal('-1') if the first
operand is lower in the total order than the second, and \texttt{Decimal('1')} if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise \texttt{InvalidOperation} if the second operand cannot be converted exactly.

\texttt{compare_total_mag} (\texttt{other}, \texttt{context}=None)

Compare two operands using their abstract representation rather than their value as in \texttt{compare_total()}, but ignoring the sign of each operand. \texttt{x.compare_total_mag(y)} is equivalent to \texttt{x.copy_abs().compare_total(y.copy_abs())}.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise \texttt{InvalidOperation} if the second operand cannot be converted exactly.

\texttt{conjugate}()

Just returns self, this method is only to comply with the Decimal Specification.

\texttt{copy_abs}()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

\texttt{copy_negate}()

Return the negation of the argument. This operation is
unaffected by the context and is quiet: no flags are changed and no rounding is performed.

**copy_sign**(other, context=None)
Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```python
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

**exp**(context=None)
Return the value of the (natural) exponential function $e^{x}$ at the given number. The result is correctly rounded using the \texttt{ROUND_HALF_EVEN} rounding mode.

```python
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

**from_float**(f)
Classmethod that converts a float to a decimal number, exactly.

Note \texttt{Decimal.from_float(0.1)} is not the same as \texttt{Decimal('0.1')}. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is $0x1.999999999999ap-4$. That equivalent value in decimal is $0.1000000000000000555111512312578270211815834045410$
**Note:** From Python 3.2 onwards, a `Decimal` instance can also be constructed directly from a `float`.

```python
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

*New in version 3.1.*

**fma**(other, third, context=None)

Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```python
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

**is_canonical()**

Return `True` if the argument is canonical and `False` otherwise. Currently, a `Decimal` instance is always canonical, so this operation always returns `True`.

**is_finite()**

Return `True` if the argument is a finite number, and `False` if the argument is an infinity or a NaN.

**is_infinite()**

Return `True` if the argument is either positive or negative
infinity and False otherwise.

is_nan()
Return True if the argument is a (quiet or signaling) NaN and False otherwise.

is_normal(context=None)
Return True if the argument is a normal finite number. Return False if the argument is zero, subnormal, infinite or a NaN.

is_qnan()
Return True if the argument is a quiet NaN, and False otherwise.

is_signed()
Return True if the argument has a negative sign and False otherwise. Note that zeros and NaNs can both carry signs.

is_snan()
Return True if the argument is a signaling NaN and False otherwise.

is_subnormal(context=None)
Return True if the argument is subnormal, and False otherwise.

is_zero()
Return True if the argument is a (positive or negative) zero and False otherwise.

ln(context=None)
Return the natural (base e) logarithm of the operand. The result
is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

**log10(context=None)**

Return the base ten logarithm of the operand. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

**logb(context=None)**

For a nonzero number, return the adjusted exponent of its operand as a `Decimal` instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the `DivisionByZero` flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

**logical_and(other, context=None)**

`logical_and()` is a logical operation which takes two `logical operands` (see `Logical operands`). The result is the digit-wise `and` of the two operands.

**logical_invert(context=None)**

`logical_invert()` is a logical operation. The result is the digit-wise inversion of the operand.

**logical_or(other, context=None)**

`logical_or()` is a logical operation which takes two `logical operands` (see `Logical operands`). The result is the digit-wise `or` of the two operands.

**logical_xor(other, context=None)**

`logical_xor()` is a logical operation which takes two `logical operands` (see `Logical operands`). The result is the digit-wise
exclusive or of the two operands.

\textbf{max}(other, context=None)

Like \texttt{max(self, other)} except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

\textbf{max_mag}(other, context=None)

Similar to the \texttt{max()} method, but the comparison is done using the absolute values of the operands.

\textbf{min}(other, context=None)

Like \texttt{min(self, other)} except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

\textbf{min_mag}(other, context=None)

Similar to the \texttt{min()} method, but the comparison is done using the absolute values of the operands.

\textbf{next_minus}(context=None)

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

\textbf{next_plus}(context=None)

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

\textbf{next_toward}(other, context=None)
If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

**normalize**(context=None)

Normalize the number by stripping the rightmost trailing zeros and converting any result equal to Decimal('0') to Decimal('0e0'). Used for producing canonical values for attributes of an equivalence class. For example, Decimal('32.100') and Decimal('0.321000e+2') both normalize to the equivalent value Decimal('32.1').

**number_class**(context=None)

Return a string describing the class of the operand. The returned value is one of the following ten strings.

- "-Infinity", indicating that the operand is negative infinity.
- "-Normal", indicating that the operand is a negative normal number.
- "-Subnormal", indicating that the operand is negative and subnormal.
- "-Zero", indicating that the operand is a negative zero.
- "+Zero", indicating that the operand is a positive zero.
- "+Subnormal", indicating that the operand is positive and subnormal.
- "+Normal", indicating that the operand is a positive normal number.
- "+Infinity", indicating that the operand is positive infinity.
- "NaN", indicating that the operand is a quiet NaN (Not a Number).
- "sNaN", indicating that the operand is a signaling NaN.

**quantize**(exp, rounding=None, context=None)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```python
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an **InvalidOperation** is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the **rounding** argument if given, else by the given **context** argument; if neither argument is given the rounding mode of the current thread's context is used.

An error is returned whenever the resulting exponent is greater than Emax or less than Etiny.

**radix()**

Return **Decimal(10)**, the radix (base) in which the **Decimal** class does all its arithmetic. Included for compatibility with the specification.
**remainder_near**(other, context=None)

Return the remainder from dividing self by other. This differs from self % other in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is self - n * other where n is the integer nearest to the exact value of self / other, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of self.

```python
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

**rotate**(other, context=None)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

**same_quantum**(other, context=None)

Test whether self and other have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are
changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

**scaleb**(other, context=None)
Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by $10^{\text{other}}$. The second operand must be an integer.

**shift**(other, context=None)
Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range $-\text{precision}$ through $\text{precision}$. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

**sqrt**(context=None)
Return the square root of the argument to full precision.

**to_eng_string**(context=None)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`. 
**to_integral**(*rounding=None, context=None*)

Identical to the **to_integral_value()** method. The **to_integral** name has been kept for compatibility with older versions.

**to_integral_exact**(*rounding=None, context=None*)

Round to the nearest integer, signaling **Inexact** or **Rounded** as appropriate if rounding occurs. The rounding mode is determined by the **rounding** parameter if given, else by the given **context**. If neither parameter is given then the rounding mode of the current context is used.

**to_integral_value**(*rounding=None, context=None*)

Round to the nearest integer without signaling **Inexact** or **Rounded**. If given, applies **rounding**; otherwise, uses the rounding method in either the supplied **context** or the current context.

**Logical operands**

The logical_and(), logical_invert(), logical_or(), and logical_xor() methods expect their arguments to be **logical operands**. A **logical operand** is a **Decimal** instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.
Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions:

```python
decimal.getcontext()
    Return the current context for the active thread.

decimal.setcontext(c)
    Set the current context for the active thread to c.
```

You can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

```python
decimal.localcontext()  # Return a context manager that will set the current context for the active thread to a copy of ctx on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.
```

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```python
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42  # Perform a high precision calculation
    s = calculate_something()
```
New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts:

**class decimal. `BasicContext`**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_UP`. All flags are cleared. All traps are enabled (treated as exceptions) except `Inexact`, `Rounded`, and `Subnormal`.

Because many of the traps are enabled, this context is useful for debugging.

**class decimal. `ExtendedContext`**

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_EVEN`. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of NaN or Infinity instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

**class decimal. `DefaultContext`**

This context is used by the `Context` constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the `Context`
constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are prec=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

In addition to the three supplied contexts, new contexts can be created with the Context constructor.

```python
class decimal.Context(prec=None, rounding=None, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None)
```

Creates a new context. If a field is not specified or is None, the default values are copied from the DefaultContext. If the flags field is not specified or is None, all flags are cleared.

prec is an integer in the range [1, MAX_PREC] that sets the precision for arithmetic operations in the context.

The rounding option is one of the constants listed in the section Rounding Modes.

The traps and flags fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.
The $E_{min}$ and $E_{max}$ fields are integers specifying the outer limits allowable for exponents. $E_{min}$ must be in the range $[\text{MIN}_E_{MIN}, 0]$, $E_{max}$ in the range $[0, \text{MAX}_E_{MAX}]$.

The $capitals$ field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: `Decimal('6.02e+23')`.

The $clamp$ field is either 0 (the default) or 1. If set to 1, the exponent $e$ of a `Decimal` instance representable in this context is strictly limited to the range $E_{min} - \text{prec} + 1 \leq e \leq E_{max} - \text{prec} + 1$. If $clamp$ is 0 then a weaker condition holds: the adjusted exponent of the `Decimal` instance is at most $E_{max}$. When $clamp$ is 1, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23000E+999')
Decimal('1.23000E+999')
```

A $clamp$ value of 1 allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The `Context` class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the `Decimal` methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding `Context` method. For example, for a `Context` instance $C$ and `Decimal`
instance $x$, $C.exp(x)$ is equivalent to $x.exp(context=C)$. Each `Context` method accepts a Python integer (an instance of `int`) anywhere that a Decimal instance is accepted.

`clear_flags()`
Resets all of the flags to 0.

`clear_traps()`
Resets all of the traps to 0.

*New in version 3.3.*

`copy()`
Return a duplicate of the context.

`copy_decimal(num)`
Return a copy of the Decimal instance `num`.

`create_decimal(num)`
Creates a new Decimal instance from `num` but using `self` as context. Unlike the `Decimal` constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```python
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
```
This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

```python
create_decimal_from_float(f)

Creates a new Decimal instance from a float f but rounding using self as the context. Unlike the Decimal.from_float() class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```python
context = Context(prec=5, rounding=ROUND_DOWN)
context.create_decimal_from_float(math.pi)
Decimal('3.1415')
context = Context(prec=5, traps=[Inexact])
context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
  ... 
decimal.Inexact: None
```

*New in version 3.1.*

**Etiny()**

Returns a value equal to $E_{\text{min}} - \text{prec} + 1$ which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to $\text{Etiny}$.

**Etop()**

Returns a value equal to $E_{\text{max}} - \text{prec} + 1$.

The usual approach to working with decimals is to create `Decimal`
instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the `Decimal` class and are only briefly recounted here.

\[ \text{abs}(x) \]

Returns the absolute value of \( x \).

\[ \text{add}(x, y) \]

Return the sum of \( x \) and \( y \).

\[ \text{canonical}(x) \]

Returns the same Decimal object \( x \).

\[ \text{compare}(x, y) \]

Compares \( x \) and \( y \) numerically.

\[ \text{compare\_signal}(x, y) \]

Compares the values of the two operands numerically.

\[ \text{compare\_total}(x, y) \]

Compares two operands using their abstract representation.

\[ \text{compare\_total\_mag}(x, y) \]

Compares two operands using their abstract representation, ignoring sign.

\[ \text{copy\_abs}(x) \]

Returns a copy of \( x \) with the sign set to 0.

\[ \text{copy\_negate}(x) \]

Returns a copy of \( x \) with the sign inverted.
**copy_sign**\((x, y)\)
Copies the sign from \(y\) to \(x\).

**divide**\((x, y)\)
Return \(x\) divided by \(y\).

**divide_int**\((x, y)\)
Return \(x\) divided by \(y\), truncated to an integer.

**divmod**\((x, y)\)
Divides two numbers and returns the integer part of the result.

**exp**\((x)\)
Returns \(e^{**x}\).

**fma**\((x, y, z)\)
Returns \(x\) multiplied by \(y\), plus \(z\).

**is_canonical**\((x)\)
Returns `True` if \(x\) is canonical; otherwise returns `False`.

**is_finite**\((x)\)
Returns `True` if \(x\) is finite; otherwise returns `False`.

**is_infinite**\((x)\)
Returns `True` if \(x\) is infinite; otherwise returns `False`.

**is_nan**\((x)\)
Returns `True` if \(x\) is a qNaN or sNaN; otherwise returns `False`.

**is_normal**\((x)\)
Returns `True` if \(x\) is a normal number; otherwise returns
False.

is_qnan(x)
Returns True if x is a quiet NaN; otherwise returns False.

is_signed(x)
Returns True if x is negative; otherwise returns False.

is_snan(x)
Returns True if x is a signaling NaN; otherwise returns False.

is_subnormal(x)
Returns True if x is subnormal; otherwise returns False.

is_zero(x)
Returns True if x is a zero; otherwise returns False.

ln(x)
Returns the natural (base e) logarithm of x.

log10(x)
Returns the base 10 logarithm of x.

logb(x)
Returns the exponent of the magnitude of the operand’s MSD.

logical_and(x, y)
Applies the logical operation and between each operand’s digits.

logical_invert(x)
Invert all the digits in x.
**logical_or**(*x*, *y*)

Applies the logical operation *or* between each operand’s digits.

**logical_xor**(*x*, *y*)

Applies the logical operation *xor* between each operand’s digits.

**max**(*x*, *y*)

Compares two values numerically and returns the maximum.

**max_mag**(*x*, *y*)

Compares the values numerically with their sign ignored.

**min**(*x*, *y*)

Compares two values numerically and returns the minimum.

**min_mag**(*x*, *y*)

Compares the values numerically with their sign ignored.

**minus**(*x*)

Minus corresponds to the unary prefix minus operator in Python.

**multiply**(*x*, *y*)

Return the product of *x* and *y*.

**next_minus**(*x*)

Returns the largest representable number smaller than *x*.

**next_plus**(*x*)

Returns the smallest representable number larger than *x*.

**next_toward**(*x*, *y*)

Returns the number closest to *x*, in direction towards *y*.
normalize\((x)\)

Reduces \(x\) to its simplest form.

number_class\((x)\)

Returns an indication of the class of \(x\).

plus\((x)\)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is not an identity operation.

power\((x, y, \text{modulo}=\text{None})\)

Return \(x\) to the power of \(y\), reduced modulo \text{modulo} if given.

With two arguments, compute \(x^{**}y\). If \(x\) is negative then \(y\) must be integral. The result will be inexact unless \(y\) is integral and the result is finite and can be expressed exactly in ‘precision’ digits. The rounding mode of the context is used. Results are always correctly-rounded in the Python version.

\textit{Changed in version 3.3}: The C module computes power() in terms of the correctly-rounded \text{exp()} and \text{ln()} functions. The result is well-defined but only “almost always correctly-rounded”.

With three arguments, compute \((x^{**}y) \% \text{modulo}\). For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- \(y\) must be nonnegative
- at least one of \(x\) or \(y\) must be nonzero
- \text{modulo} must be nonzero and have at most ‘precision’
The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y`, and `modulo`. The result is always exact.

`quantize(x, y)`

Returns a value equal to `x` (rounded), having the exponent of `y`.

`radix()`

Just returns 10, as this is Decimal, :)

`remainder(x, y)`

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

`remainder_near(x, y)`

Returns `x - y * n`, where `n` is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of `x`).

`rotate(x, y)`

Returns a rotated copy of `x`, `y` times.

`same_quantum(x, y)`

Returns `True` if the two operands have the same exponent.

`scaleb(x, y)`
Returns the first operand after adding the second value its exp.

**shift**(x, y)
Returns a shifted copy of x, y times.

**sqrt**(x)
Square root of a non-negative number to context precision.

**subtract**(x, y)
Return the difference between x and y.

**to_eng_string**(x)
Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

**to_integral_exact**(x)
Rounds to an integer.

**to_sci_string**(x)
Converts a number to a string using scientific notation.
### Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal.MAX_PREC</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>decimal.MAX_EMAX</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>decimal.MIN_EMIN</td>
<td>-4250000000</td>
<td>-999999999999999999</td>
</tr>
<tr>
<td>decimal.MIN_ETINY</td>
<td>-849999999</td>
<td>-199999999999999999</td>
</tr>
</tbody>
</table>

**decimal.HAVE_THREADS**

The default value is **True**. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is **False**.
Rounding modes

decimal. **ROUND_CEILING**
Round towards Infinity.

decimal. **ROUND_DOWN**
Round towards zero.

decimal. **ROUND_FLOOR**
Round towards -Infinity.

decimal. **ROUND_HALF_DOWN**
Round to nearest with ties going towards zero.

decimal. **ROUND_HALF_EVEN**
Round to nearest with ties going to nearest even integer.

decimal. **ROUND_HALF_UP**
Round to nearest with ties going away from zero.

decimal. **ROUND_UP**
Round away from zero.

decimal. **ROUND_05UP**
Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.
Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the `DivisionByZero` trap is set, then a `DivisionByZero` exception is raised upon encountering the condition.

```python
class decimal.Clamped
    Altered an exponent to fit representation constraints.

    Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.
```

```python
class decimal.DecimalException
    Base class for other signals and a subclass of `ArithmeticError`.
```

```python
class decimal.DivisionByZero
    Signals the division of a non-infinite number by zero.

    Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity
```
or -Infinity with the sign determined by the inputs to the calculation.

**class decimal.Inexact**

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

**class decimal.InvalidOperation**

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

**class decimal.Overflow**

Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, Inexact and Rounded are also signaled.
class decimal. **Rounded**

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal. **Subnormal**

Exponent was lower than Emin prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal. **Underflow**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. **Inexact** and **Subnormal** are also signaled.

class decimal. **FloatOperation**

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the `Decimal` constructor, `create_decimal()` and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting `FloatOperation` in the context flags. Explicit conversions with `from_float()` or `create_decimal_from_float()` do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise
The following table summarizes the hierarchy of signals:

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>exceptions.ArithmeticError(exceptions.Exception)</td>
</tr>
<tr>
<td>DecimalException</td>
</tr>
<tr>
<td>Clamped</td>
</tr>
<tr>
<td>DivisionByZero(DecimalException, exceptions.)</td>
</tr>
<tr>
<td>Inexact</td>
</tr>
<tr>
<td>Overflow(Inexact, Rounded)</td>
</tr>
<tr>
<td>Underflow(Inexact, Rounded, Subnormal)</td>
</tr>
<tr>
<td>InvalidOperation</td>
</tr>
<tr>
<td>Rounded</td>
</tr>
<tr>
<td>Subnormal</td>
</tr>
<tr>
<td>FloatOperation(DecimalException, exceptions.)</td>
</tr>
</tbody>
</table>
Floating Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent $0.1$ exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```python
# Examples from Seminumerical Algorithms, Section 4.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal('11111113'), Decimal('-11111111'),
>>> (u + v) + w  # Decimal('9.5111111')
>>> u + (v + w)  # Decimal('10')

>>> u, v, w = Decimal('20000'), Decimal('-6'), Decimal('6.0000003')
>>> (u*v) + (u*w)  # Decimal('0.01')
>>> u * (v+w)  # Decimal('0.0060000')
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:
Special values

The number system for the `decimal` module provides special values including NaN, sNaN, -Infinity, Infinity, and two zeros, +0 and -0.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return NaN, or if the `InvalidOperation` signal is trapped, raise an exception. For
example, $0/0$ returns NaN which means “not a number”. This variety of NaN is quiet and, once created, will flow through other computations always resulting in another NaN. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is sNaN which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a NaN is involved. A test for equality where one of the operands is a quiet or signaling NaN always returns False (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns True. An attempt to compare two Decimals using any of the `<`, `<=`, `>` or `>=` operators will raise the `InvalidOperation` signal if either operand is a NaN, and return False if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a NaN were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the `compare()` and `compare-signal()` methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there
are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```python
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```
Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done before any threads are started so that there won’t be a race condition between threads calling `getcontext()`. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
    t1.start()
t2.start()
t3.start()
    ...
```
Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```python
def moneyfmt(value, places=2, curr='', sep='.', dp=' ', pos='', neg='-', trailneg=' '):
    """Convert Decimal to a money formatted string.

    places:    required number of places after the decimal point
    curr:     optional currency symbol before the sign
    sep:      optional grouping separator (comma, period, space, or blank)
    dp:       decimal point indicator (comma or period)
              only specify as blank when places is zero
    pos:      optional sign for positive numbers: '+'
    neg:      optional sign for negative numbers: '-'
    trailneg: optional trailing minus indicator: '-'

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg=''>
    '<0.02>'
    ""

    q = Decimal(10) ** -places        # 2 places --> 0.01
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
build, next = result.append, digits.pop
if sign:
    build(trailneg)
for i in range(places):
    build(next() if digits else '0')
if places:
    build(dp)
if not digits:
    build('0')
i = 0
while digits:
    build(next())
    i += 1
    if i == 3 and digits:
        i = 0
        build(sep)
build(curr)
built(neg if sign else pos)
return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision."
    
    >>> print(pi())
    3.141592653589793238462643383
    
    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0"
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
def exp(x):
   """Return e raised to the power of x. Result type matches input type.
   >>> print(exp(Decimal(1)))
   2.718281828459045235360287471
   >>> print(exp(Decimal(2)))
   7.389056098930650227230427461
   >>> print(exp(2.0))
   7.38905609893
   >>> print(exp(2+0j))
   (7.38905609893+0j)
   """
   getcontext().prec += 2
   i, lasts, s, fact, num = 0, 0, 1, 1, 1
   while s != lasts:
       lasts = s
       i += 1
       fact *= i
       num *= x
       s += num / fact
   getcontext().prec -= 2
   return +s

def cos(x):
   """Return the cosine of x as measured in radians.
   The Taylor series approximation works best for a small value of x.
   For larger values, first compute x = x % (2 * pi).
   >>> print(cos(Decimal('0.5')))  
   0.8775825618903727161162815826
   >>> print(cos(0.5))
   0.87758256189
   >>> print(cos(0.5+0j))
   (0.8775825618903727161162815826+0j)
(0.87758256189+0j)

"""
getcontext().prec += 2
i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi)
    """
    >>> print(sin(Decimal('0.5')))
    0.479425538604203002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

"""
s += num / fact * sign
getcontext().prec -= 2
return +s
Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```python
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the `Inexact` trap is set, it is also useful for validation:

```python
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')

>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context)
Decimal('3.21')

>>> Decimal('3.214').quantize(TWOPLACES, context=Context)
Traceback (most recent call last):
...
```
Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a quantize() step:

```python
>>> a = Decimal('102.72')  # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b  # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42  # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)  # Must quantize
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)  # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the quantize() step:

```python
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)

>>> mul(a, b)  # Automatically preserved
Decimal('325.62')
>>> div(b, a)
```
Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and 02E+4 all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The normalize() method maps all equivalent values to a single representative:

```python
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split)
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing 5.0E+3 as 5000 keeps the value constant but cannot show the original’s two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```python
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral

>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```
Q. Is there a way to convert a regular float to a Decimal?

A. Yes, any binary floating point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```python
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that “what you type is what you get”. A disadvantage is that the results can look odd if you forget that the inputs haven’t been rounded:

```python
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2
```
The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```python
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')  # unary plus trigger
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method:

```python
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345')
Decimal('1.2345')
```

Q. Is the CPython implementation fast for large numbers?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed `libmpdec` library for arbitrary precision correctly-rounded decimal floating point arithmetic. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers. However, to realize this performance gain, the context needs to be set for unrounded calculations.

```python
>>> c = getcontext()
>>> c.prec = MAX_PREC
>>> c.Emax = MAX_EMAX
>>> c.Emin = MIN_EMIN
```

*New in version 3.3.*
The `fractions` module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

```python
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

The first version requires that `numerator` and `denominator` are instances of `numbers.Rational` and returns a new `Fraction` instance with value `numerator/denominator`. If `denominator` is 0, it raises a `ZeroDivisionError`. The second version requires that `other_fraction` is an instance of `numbers.Rational` and returns a `Fraction` instance with the same value. The next two versions accept either a `float` or a `decimal.Decimal` instance, and return a `Fraction` instance with exactly the same value. Note that due to the usual issues with binary floating-point (see Floating Point Arithmetic: Issues and Limitations), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does not return `Fraction(11, 10)` as one might expect. (But see the documentation for the `limit_denominator()` method below.) The last version of the
constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator [\'/\' denominator]
```

where the optional `sign` may be either `+` or `-` and `numerator` and `denominator` (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the `float` constructor is also accepted by the `Fraction` constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('- .125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 10000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
```
The `Fraction` class inherits from the abstract base class `numbers.Rational`, and implements all of the methods and operations from that class. `Fraction` instances are hashable, and should be treated as immutable. In addition, `Fraction` has the following properties and methods:

*Changed in version 3.2:* The `Fraction` constructor now accepts `float` and `decimal.Decimal` instances.

- **numerator**
  Numerator of the `Fraction` in lowest term.

- **denominator**
  Denominator of the `Fraction` in lowest term.

- **from_float(flt)**
  This class method constructs a `Fraction` representing the exact value of `flt`, which must be a `float`. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

  **Note:** From Python 3.2 onwards, you can also construct a `Fraction` instance directly from a `float`.

- **from_decimal(dec)**
  This class method constructs a `Fraction` representing the exact value of `dec`, which must be a `decimal.Decimal` instance.

  **Note:** From Python 3.2 onwards, you can also construct a
**Fraction** instance directly from a **decimal.Decimal** instance.

**limit_denominator**(max_denominator=1000000)
Finds and returns the closest **Fraction** to self that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```python
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator
Fraction(355, 113)
```
or for recovering a rational number that's represented as a float:

```python
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()
Returns the greatest `int` <= self. This method can also be accessed through the **math.floor()** function:

```python
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()
Returns the least `int` >= self. This method can also be
accessed through the `math.ceil()` function.

```python
__round__()
__round__(ndigits)
```

The first version returns the nearest `int` to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the `round()` function.

`fractions.gcd(a, b)`

Return the greatest common divisor of the integers `a` and `b`. If either `a` or `b` is nonzero, then the absolute value of `gcd(a, b)` is the largest integer that divides both `a` and `b`. `gcd(a, b)` has the same sign as `b` if `b` is nonzero; otherwise it takes the sign of `a`. `gcd(0, 0)` returns `0`.

*Deprecated since version 3.5:* Use `math.gcd()` instead.

**See also:**

**Module numbers**

The abstract base classes making up the numeric tower.
random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range \([0.0, 1.0)\). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of \(2^{19937}-1\). The underlying implementation in C is both fast and thread safe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate
your own instances of Random to get generators that don’t share state.

Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), and setstate() methods. Optionally, a new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range.

The random module also provides the SystemRandom class which uses the system function os.urandom() to generate random numbers from sources provided by the operating system.

**Warning:** The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the secrets module.


Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.
Bookkeeping functions

random.seed(a=None, version=2)

Initialize the random number generator.

If a is omitted or None, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the os.urandom() function for details on availability).

If a is an int, it is used directly.

With version 2 (the default), a str, bytes, or bytearray object gets converted to an int and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for str and bytes generates a narrower range of seeds.

*Changed in version 3.2:* Moved to the version 2 scheme which uses all of the bits in a string seed.

random.getstate()

Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state.

random.setstate(state)

state should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time getstate() was called.
random.\texttt{getrandbits(k)}

Returns a Python integer with $k$ random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, \texttt{getrandbits()} enables \texttt{randrange()} to handle arbitrarily large ranges.
Functions for integers

```python
random.randrange(stop)
random.randrange(start, stop[, step])
```

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn’t actually build a range object.

The positional argument pattern matches that of `range()`. Keyword arguments should not be used because the function may use them in unexpected ways.

*Changed in version 3.2:* `randrange()` is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random() * n)` which could produce slightly uneven distributions.

```python
random.randint(a, b)
```

Return a random integer \( N \) such that \( a \leq N \leq b \). Alias for `randrange(a, b+1)`.
Functions for sequences

random.choice(seq)
Return a random element from the non-empty sequence seq. If seq is empty, raises IndexError.

random.choices(population, weights=None, *, cum_weights=None, k=1)
Return a k sized list of elements chosen from the population with replacement. If the population is empty, raises IndexError.

If a weights sequence is specified, selections are made according to the relative weights. Alternatively, if a cum_weights sequence is given, the selections are made according to the cumulative weights (perhaps computed using itertools.accumulate()). For example, the relative weights [10, 5, 30, 5] are equivalent to the cumulative weights [10, 15, 45, 50]. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither weights nor cum_weights are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the population sequence. It is a TypeError to specify both weights and cum_weights.

The weights or cum_weights can use any numeric type that interoperates with the float values returned by random() (that includes integers, floats, and fractions but excludes decimals).

For a given seed, the choices() function with equal weighting typically produces a different sequence than repeated calls to
choice(). The algorithm used by choices() uses floating point arithmetic for internal consistency and speed. The algorithm used by choice() defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

New in version 3.6.

random.shuffle(x[, random])
Shuffle the sequence x in place.

The optional argument random is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function random().

To shuffle an immutable sequence and return a new shuffled list, use sample(x, k=len(x)) instead.

Note that even for small len(x), the total number of permutations of x can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

random.sample(population, k)
Return a k length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).
Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use a `range()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a `ValueError` is raised.
Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution’s equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random. `random()`
Return the next random floating point number in the range [0.0, 1.0).

random. `uniform(a, b)`
Return a random floating point number $N$ such that $a \leq N \leq b$ for $a \leq b$ and $b \leq N \leq a$ for $b < a$.

The end-point value $b$ may or may not be included in the range depending on floating-point rounding in the equation $a + (b-a) \times \text{random()}$.

random. `triangular(low, high, mode)`
Return a random floating point number $N$ such that $\text{low} \leq N \leq \text{high}$ and with the specified $mode$ between those bounds. The low and high bounds default to zero and one. The mode argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random. `betavariate(alpha, beta)`
Beta distribution. Conditions on the parameters are $\alpha > 0$ and $\beta > 0$. Returned values range between 0 and 1.

random. `expovariate(lambd)`
Exponential distribution. $\lambda$ is $1.0$ divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if $\lambda$ is positive, and from negative infinity to 0 if $\lambda$ is negative.

```python
random.gammavariate(alpha, beta)
```
Gamma distribution. (Not the gamma function!) Conditions on the parameters are $\alpha > 0$ and $\beta > 0$.

The probability distribution function is:

$$pdf(x) = \frac{x^{(\alpha - 1)} \cdot \exp(-x / \beta)}{\Gamma(\alpha) \cdot \beta^\alpha}$$

```python
random.gauss(mu, sigma)
```
Gaussian distribution. $\mu$ is the mean, and $\sigma$ is the standard deviation. This is slightly faster than the `normalvariate()` function defined below.

```python
random.lognormvariate(mu, sigma)
```
Log normal distribution. If you take the natural logarithm of this distribution, you’ll get a normal distribution with mean $\mu$ and standard deviation $\sigma$. $\mu$ can have any value, and $\sigma$ must be greater than zero.

```python
random.normalvariate(mu, sigma)
```
Normal distribution. $\mu$ is the mean, and $\sigma$ is the standard deviation.

```python
random.vonmisesvariate(mu, kappa)
```
$\mu$ is the mean angle, expressed in radians between 0 and $2\pi$,.
and \(\kappa\) is the concentration parameter, which must be greater than or equal to zero. If \(\kappa\) is equal to zero, this distribution reduces to a uniform random angle over the range 0 to \(2\pi\).

random. \texttt{pareto} \texttt{variate}(\alpha)

Pareto distribution. \(\alpha\) is the shape parameter.

random. \texttt{weibull} \texttt{variate}(\alpha, \beta)

Weibull distribution. \(\alpha\) is the scale parameter and \(\beta\) is the shape parameter.
Alternative Generator

```
class random.SystemRandom([seed])
```

Class that uses the `os.urandom()` function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the `seed()` method has no effect and is ignored. The `getstate()` and `setstate()` methods raise `NotImplementedError` if called.
Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator’s `random()` method will continue to produce the same sequence when the compatible seeder is given the same seed.
Examples and Recipes

Basic examples:

```python
>>> random()  # Random float
0.37444887175646646

>>> uniform(2.5, 10.0)  # Random float
3.1800146073117523

>>> expovariate(1 / 5)  # Interval between arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)  # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)  # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])  # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()  # Shuffle a list
>>> shuffle(deck)
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)  # Four samples without replacement
[40, 10, 50, 30]
```

Simulations:

```python
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices([['red', 'black', 'green'], [18, 18, 2]], k=6)
```
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more
>>> # of a biased coin that settles on heads 60% of
>>> def trial():
...    return choices('HT', cum_weights=(0.60, 1.00))
...
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in
>>> def trial():
...    return 2500 <= sorted(choices(range(10000)),
...
>>> sum(trial() for i in range(10000)) / 10000
0.7958

Example of statistical bootstrapping using resampling with
replacement to estimate a confidence interval for the mean of a
sample of size five:

# http://statistics.about.com/od/Applications/a/Exan
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in ran
print(f'The sample mean of {mean(data):.1f} has a 90
Example of a resampling permutation test to determine the statistical significance or p-value of an observed difference between the effects of a drug versus a placebo:

```python
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} that at least as extreme as the observed difference.
The one-sided p-value of {count / n:.4f} leads us to reject the null hypothesis that there is no difference between the drug and the placebo.
```

Simulation of arrival times and service deliveries in a single server queue:

```python
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
```
```python
stdev_service_time = 0.5
num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else service_time
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}. Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}. Max wait: {max(waits):.1f}.')

See also: Statistics for Hackers a video tutorial by Jake Vanderplas on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.
statistics — Mathematical statistics functions

New in version 3.4.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.

Note: Unless explicitly noted otherwise, these functions support \texttt{int}, \texttt{float}, \texttt{decimal.Decimal} and \texttt{fractions.Fraction}. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Mixed types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use \texttt{map()} to ensure a consistent result, e.g. \texttt{map(float, input_data)}. 
Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean()</td>
<td>Arithmetic mean (&quot;average&quot;) of data.</td>
</tr>
<tr>
<td>harmonic_mean()</td>
<td>Harmonic mean of data.</td>
</tr>
<tr>
<td>median()</td>
<td>Median (middle value) of data.</td>
</tr>
<tr>
<td>median_low()</td>
<td>Low median of data.</td>
</tr>
<tr>
<td>median_high()</td>
<td>High median of data.</td>
</tr>
<tr>
<td>median_grouped()</td>
<td>Median, or 50th percentile, of grouped data.</td>
</tr>
<tr>
<td>mode()</td>
<td>Mode (most common value) of discrete data.</td>
</tr>
</tbody>
</table>
Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pstdev()</td>
<td>Population standard deviation of data.</td>
</tr>
<tr>
<td>pvariance()</td>
<td>Population variance of data.</td>
</tr>
<tr>
<td>stdev()</td>
<td>Sample standard deviation of data.</td>
</tr>
<tr>
<td>variance()</td>
<td>Sample variance of data.</td>
</tr>
</tbody>
</table>
Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

**statistics.mean(data)**

Return the sample arithmetic mean of *data* which can be a sequence or iterator.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called “the average”, although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, *StatisticsError* will be raised.

Some examples of use:

```python
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.3")])
Decimal('0.5625')
```

**Note:** The mean is strongly affected by outliers and is not a
A robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust, although less efficient, measures of central location, see \texttt{median()} and \texttt{mode()}. (In this case, “efficient” refers to statistical efficiency rather than computational efficiency.) The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, \( \text{mean(sample)} \) converges on the true mean of the entire population. If \( \text{data} \) represents the entire population rather than a sample, then \( \text{mean(data)} \) is equivalent to calculating the true population mean \( \mu \).

\textbf{statistics.} \texttt{harmonic\_mean(data)}

Return the harmonic mean of \( \text{data} \), a sequence or iterator of real-valued numbers.

The harmonic mean, sometimes called the subcontrary mean, is the reciprocal of the arithmetic \texttt{mean()} of the reciprocals of the data. For example, the harmonic mean of three values \( a, b \) and \( c \) will be equivalent to \( \frac{3}{\left(\frac{1}{a} + \frac{1}{b} + \frac{1}{c}\right)} \).

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging quantities which are rates or ratios, for example speeds. For example:

Suppose an investor purchases an equal value of shares in each of three companies, with P/E (price/earning) ratios of 2.5, 3 and 10. What is the average P/E ratio for the investor’s portfolio?

\begin{verbatim}
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment
3.6
\end{verbatim}
Using the arithmetic mean would give an average of about 5.167, which is too high.

`StatisticsError` is raised if `data` is empty, or any element is less than zero.

*New in version 3.6.*

```
statistics.median(data)
```

Return the median (middle value) of numeric data, using the common “mean of middle two” method. If `data` is empty, `StatisticsError` is raised. `data` can be a sequence or iterator.

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don’t mind that the median may not be an actual data point.

If your data is ordinal (supports order operations) but not numeric (doesn’t support addition), you should use `median_low()` or `median_high()` instead.
```
See also: median_low(), median_high(), median_grouped()
```

**statistics.median_low(data)**

Return the low median of numeric data. If `data` is empty, `StatisticsError` is raised. `data` can be a sequence or iterator.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

**statistics.median_high(data)**

Return the high median of data. If `data` is empty, `StatisticsError` is raised. `data` can be a sequence or iterator.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer
the median to be an actual data point rather than interpolated.

**statistics.median_grouped**(data, interval=1)

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If data is empty, **StatisticsError** is raised. data can be a sequence or iterator.

```python
>>> median_grouped([52, 52, 53, 54])
52.5
```

In the following example, the data are rounded, so that each value represents the midpoint of data classes, e.g. 1 is the midpoint of the class 0.5–1.5, 2 is the midpoint of 1.5–2.5, 3 is the midpoint of 2.5–3.5, etc. With the data given, the middle value falls somewhere in the class 3.5–4.5, and interpolation is used to estimate it:

```python
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

Optional argument *interval* represents the class interval, and defaults to 1. Changing the class interval naturally will change the interpolation:

```python
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

This function does not check whether the data points are at least *interval* apart.

**CPython implementation detail:** Under some circumstances, **median_grouped()** may coerce data points to floats. This
behaviour is likely to change in the future.

See also:
- The SSMEDIAN function in the Gnome Gnumeric spreadsheet, including this discussion.

statistics.mode(data)

Return the most common data point from discrete or nominal data. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If data is empty, or if there is not exactly one most common value, StatisticsError is raised.

mode assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```python
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data:

```python
>>> mode(['red', 'blue', 'blue', 'red', 'green', 'red'])
'red'
```

statistics.pstdev(data, mu=None)

Return the population standard deviation (the square root of the population variance). See pvariance() for arguments and other details.
```python
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

```
statistics.pvariance(data, mu=None)

Return the population variance of data, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument mu is given, it should be the mean of data. If it is missing or None (the default), the mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the variance() function is usually a better choice.

Raises StatisticsError if data is empty.

Examples:

```python
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.7
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument mu to avoid recalculation:

```python
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```
This function does not attempt to verify that you have passed the actual mean as \( mu \). Using arbitrary values for \( mu \) may lead to invalid or impossible results.

Decimals and Fractions are supported:

```python
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), Decimal('24.815')]

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

**Note:** When called with the entire population, this gives the population variance \( \sigma^2 \). When called on a sample instead, this is the biased sample variance \( s^2 \), also known as variance with \( N \) degrees of freedom.

If you somehow know the true population mean \( \mu \), you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

```
statistics.stdev(data, xbar=None)
```

Return the sample standard deviation (the square root of the sample variance). See `variance()` for arguments and other details.

```python
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```
statistics. **variance**(data, xbar=None)

Return the sample variance of data, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument xbar is given, it should be the mean of data. If it is missing or None (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see pvariance().

Raises **StatisticsError** if data has fewer than two values.

**Examples:**

```python
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the mean of your data, you can pass it as the optional second argument xbar to avoid recalculation:

```python
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as xbar. Using arbitrary values for xbar can lead to invalid or impossible results.
Decimal and Fraction values are supported:

```python
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("31.01875")])
Decimal('31.01875')
```

```python
>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

**Note:** This is the sample variance $s^2$ with Bessel’s correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean $\mu$ you should pass it to the `pvariance()` function as the $mu$ parameter to get the variance of a sample.
Exceptions

A single exception is defined:

```
exception statistics.StatisticsError
```

Subclass of `ValueError` for statistics-related exceptions.
Functional Programming Modules

The modules described in this chapter provide functions and classes that support a functional programming style, and general operations on callables.

The following modules are documented in this chapter:

- **itertools** — Functions creating iterators for efficient looping
  - Itertool functions
  - Itertools Recipes
- **functools** — Higher-order functions and operations on callable objects
  - partial Objects
- **operator** — Standard operators as functions
  - Mapping Operators to Functions
  - Inplace Operators
**itertools — Functions creating iterators for efficient looping**

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: \( \text{tabulate}(f) \) which produces a sequence \( f(0), f(1), \ldots \). The same effect can be achieved in Python by combining \( \text{map}(f) \) and \( \text{count()} \) to form \( \text{map}(f, \text{count()}) \).

These tools and their built-in counterparts also work well with the high-speed functions in the `operator` module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: \( \text{sum}(\text{map}(\text{operator.mul, vector1, vector2})) \).

**Infinite iterators:**

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>start, [step]</td>
<td>start, start+step, start+2*step, …</td>
<td><code>count(10) --&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 11 12 13 14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p0, p1, ... plast,</td>
<td><code>cycle('ABCD')</code></td>
</tr>
<tr>
<td>Function</td>
<td>Arguments</td>
<td>Results</td>
<td>Example</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>----------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>cycle()</td>
<td>p</td>
<td>p0, p1, ...</td>
<td>(--\rightarrow A \ B \ C \ D \ A \ B \ C \ D \ ...))</td>
</tr>
<tr>
<td>repeat()</td>
<td>elem [,n]</td>
<td>elem, elem, elem, ... endlessly or up to n times</td>
<td>(\text{repeat(10, 3)} \rightarrow 10 \ 10 \ 10)</td>
</tr>
</tbody>
</table>

Iterators terminating on the shortest input sequence:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
<th>Ex</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate()</td>
<td>p [,func]</td>
<td>p0, p0+p1, p0+p1+p2, ...</td>
<td>ac1</td>
</tr>
<tr>
<td>chain()</td>
<td>p, q, ...</td>
<td>p0, p1, ... plast, q0, q1, ...</td>
<td>chC</td>
</tr>
<tr>
<td>chain.from_iterable()</td>
<td>iterable</td>
<td>p0, p1, ... plast, q0, q1, ...</td>
<td>ch'D</td>
</tr>
<tr>
<td>compress()</td>
<td>data, selectors</td>
<td>(d[0] if s[0]), (d[1] if s[1]), ...</td>
<td>co1</td>
</tr>
<tr>
<td>dropwhile()</td>
<td>pred, seq</td>
<td>seq[n], seq[n+1], starting when pred fails</td>
<td>dr1</td>
</tr>
<tr>
<td>filterfalse()</td>
<td>pred, seq</td>
<td>elements of seq where pred(elem) is false</td>
<td>fi</td>
</tr>
<tr>
<td>groupby()</td>
<td>iterable[, key]</td>
<td>sub-iterators grouped by value of key(v)</td>
<td></td>
</tr>
<tr>
<td>islice()</td>
<td>seq, [start[, stop[, step]]]</td>
<td>elements from seq[start:stop:step]</td>
<td>is-</td>
</tr>
<tr>
<td>starmap()</td>
<td>func, seq</td>
<td>func(*seq[0]), func(*seq[1]), ...</td>
<td>sta</td>
</tr>
</tbody>
</table>
**Combinatoric iterators:**

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>product()</code></td>
<td>p, q, ... [repeat=1]</td>
<td>cartesian product, equivalent to a nested for-loop</td>
</tr>
<tr>
<td><code>permutations()</code></td>
<td>p[, r]</td>
<td>r-length tuples, all possible orderings, no repeated elements</td>
</tr>
<tr>
<td><code>combinations()</code></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, no repeated elements</td>
</tr>
<tr>
<td><code>combinations_withReplacement()</code></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, repeated elements possible</td>
</tr>
</tbody>
</table>
product('ABCD', repeat=2)

permutations('ABCD', 2)

combinations('ABCD', 2)

combinations_with_replacement('ABCD', 2)
Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

`itertools.accumulate(iterable[, func])`

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional `func` argument). If `func` is supplied, it should be a function of two arguments. Elements of the input `iterable` may be any type that can be accepted as arguments to `func`. (For example, with the default operation of addition, elements may be any addable type including `Decimal` or `Fraction`.) If the input iterable is empty, the output iterable will also be empty.

Roughly equivalent to:

```python
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
    yield total
```
There are a number of uses for the `func` argument. It can be set to `min()` for a running minimum, `max()` for a running maximum, or `operator.mul()` for a running product. Amortization tables can be built by accumulating interest and applying payments. First-order recurrence relations can be modeled by supplying the initial value in the iterable and using only the accumulated total in `func` argument:

```python
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))   # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))            # running maximum
[3, 4, 6, 6, 9, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001
```

See `functools.reduce()` for a similar function that returns only the final accumulated value.

*New in version 3.2.*
**Changed in version 3.3:** Added the optional `func` parameter.

`itertools.chain(*iterables)`

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Roughly equivalent to:

```python
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

**classmethod chain.from_iterable(iterable)**

Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```python
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B
    for it in iterables:
        for element in it:
            yield element
```

`itertools.combinations(iterable, r)`

Return `r` length subsequences of elements from the input `iterable`.

Combinations are emitted in lexicographic sort order. So, if the input `iterable` is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their
value. So if the input elements are unique, there will be no repeat values in each combination.

Roughly equivalent to:

```python
def combinations(iterable, r):
    pool = tuple(iterable)
n = len(pool)
if r > n:
    return
indices = list(range(r))
yield tuple(pool[i] for i in indices)
while True:
    for i in reversed(range(r)):
        if indices[i] != i + n - r:
            break
    else:
        return
    indices[i] += 1
    for j in range(i+1, r):
        indices[j] = indices[j-1] + 1
    yield tuple(pool[i] for i in indices)
```

The code for `combinations()` can be also expressed as a subsequence of `permutations()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```python
def combinations(iterable, r):
    pool = tuple(iterable)
n = len(pool)
for indices in permutations(range(n), r):
    if sorted(indices) == list(indices):
```
The number of items returned is \(\frac{n!}{r! \cdot (n-r)!}\) when \(0 \leq r \leq n\) or zero when \(r > n\).

**itertools.combinations_with_replacement**(iterable, r)

Return \(r\) length subsequences of elements from the input iterable allowing individual elements to be repeated more than once.

Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```python
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
    yield tuple(pool[i] for i in indices)
```
The code for `combinations_with_replacement()` can be also expressed as a subsequence of `product()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```python
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is \( \frac{(n+r-1)!}{r! (n-1)!} \) when \( n > 0 \).

*New in version 3.1.*

`itertools.compress(data, selectors)`

Make an iterator that filters elements from `data` returning only those that have a corresponding element in `selectors` that evaluates to `True`. Stops when either the `data` or `selectors` iterables has been exhausted. Roughly equivalent to:

```python
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E
    return (d for d, s in zip(data, selectors) if s)
```

*New in version 3.1.*

`itertools.count(start=0, step=1)`

Make an iterator that returns evenly spaced values starting with
number \textit{start}. Often used as an argument to \texttt{map()} to generate consecutive data points. Also, used with \texttt{zip()} to add sequence numbers. Roughly equivalent to:

\begin{verbatim}
def count(start=0, step=1):
    # count(10) --\rightarrow 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
\end{verbatim}

When counting with floating point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: (\texttt{start + step \times i for i in count()}).

\textit{Changed in version 3.1: Added step argument and allowed non-integer arguments.}

\texttt{itertools.cycle(iterable)}

Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

\begin{verbatim}
def cycle(iterable):
    # cycle('ABCD') --\rightarrow A B C D A B C D A B C D .
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
\end{verbatim}
Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

**itertools.** **dropwhile**(predicate, iterable)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time. Roughly equivalent to:

```python
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
        break
    for x in iterable:
        yield x
```

**itertools.** **filterfalse**(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is `None`, return the items that are false. Roughly equivalent to:

```python
def filterfalse(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```
**itertools.** `[groupby](https://docs.python.org/3/library/itertools.html#itertools.groupby)***(iterable, key=None)***

Make an iterator that returns consecutive keys and groups from the `iterable`. The `key` is a function computing a key value for each element. If not specified or is `None`, `key` defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of `groupby()` is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with `groupby()`. Because the source is shared, when the `groupby()` object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```python
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))  # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` is roughly equivalent to:

```python
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')]
    # [list(g) for k, g in groupby('AAAAABBBCCD')]
def __init__(self, iterable, key=None):
    if key is None:
```
key = lambda x: x
self.keyfunc = key
self.it = iter(iterable)
self.tgtkey = self.currkey = self.currval

def __iter__(self):
    return self
def __next__(self):
    self.id = object()
    while self.currkey == self.tgtkey:
        self.currvalue = next(self.it)  # E.
        self.currkey = self.keyfunc(self.currvalue)
    self.tgtkey = self.currkey
    return (self.currkey, self._grouper(self.

def _grouper(self, tgtkey, id):
    while self.id is id and self.currkey == t
        yield self.currvalue
    try:
        self.currvalue = next(self.it)
    except StopIteration:
        return
    self.currkey = self.keyfunc(self.currvalue)

itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])

Make an iterator that returns selected elements from the iterable. If start is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless step is set higher than one which results in items being skipped. If stop is None, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, islice() does not support negative values for start, stop, or step. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Roughly
equivdeent to:

```python
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or s.stop, s.step
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
    return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
            nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

If `start` is `None`, then iteration starts at zero. If `step` is `None`, then the step defaults to one.

`itertools.permutations(iterable, r=None)`

Return successive `r` length permutations of elements in the `iterable`.

If `r` is not specified or is `None`, then `r` defaults to the length of the
iterable and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input iterable is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Roughly equivalent to:

```python
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
n = len(pool)
r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
cycles = list(range(n, n-r, -1))
yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[:i]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
yield tuple(pool[i] for i in indices)
        break
    else:
        return
```
The code for `permutations()` can be also expressed as a subsequence of `product()`, filtered to exclude entries with repeated elements (those from the same position in the input pool):

```python
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

The number of items returned is $n! / (n-r)!$ when $0 \leq r \leq n$ or zero when $r > n$.

`itertools.product(*iterables, repeat=1)`

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` returns the same as `((x, y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional `repeat` keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`. 
This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```python
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy
    # product(range(2), repeat=3) --> 000 001 010
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

`itertools.repeat(object[, times])`

Make an iterator that returns `object` over and over again. Runs indefinitely unless the `times` argument is specified. Used as argument to `map()` for invariant parameters to the called function. Also used with `zip()` to create an invariant part of a tuple record.

Roughly equivalent to:

```python
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for `repeat` is to supply a stream of constant values to `map` or `zip`:
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**itertools.starmap(function, iterable)**

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of `map()` when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between `map()` and `starmap()` parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```python
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32
    for args in iterable:
        yield function(*args)
```

**itertools.takewhile(predicate, iterable)**

Make an iterator that returns elements from the iterable as long as the predicate is true. Roughly equivalent to:

```python
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

**itertools.tee(iterable, n=2)**

Return $n$ independent iterators from a single iterable.

The following Python code helps explain what `tee` does (although
the actual implementation is more complex and uses only a single underlying FIFO queue).

Roughly equivalent to:

```python
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:  # when the local deque is empty
                try:
                    newval = next(it)  # fetch a new value
                except StopIteration:
                    return
                for d in deques:
                    # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once `tee()` has made a split, the original `iterable` should not be used anywhere else; otherwise, the `iterable` could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

`itertools.zip_longest(*iterables, fillvalue=None)`

Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with `fillvalue`. Iteration continues until the longest iterable is exhausted. Roughly equivalent to:
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='--') -->
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
            iterators[i] = repeat(fillvalue)
            value = fillvalue
            values.append(value)
        yield tuple(values)

If one of the iterables is potentially infinite, then the zip_longest() function should be wrapped with something that limits the number of calls (for example islice() or takewhile()). If not specified, fillvalue defaults to None.
Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring “vectorized” building blocks over the use of for-loops and generators which incur interpreter overhead.

```python
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([[value], iterator])

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --&gt; E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, the iterator is consumed completely."
```

# Use functions that consume iterators at C speed

```python
if n is None:
    # feed the entire iterator into a zero-length deque
    collections.deque(iterator, maxlen=0)
else:
    # advance to the empty slice starting at position n
    next(islice(iterator, n, n), None)
```

```python
def nth(iterable, n, default=None):
    """Returns the nth item or a default value""
    return next(islice(iterable, n, n), default)
```

```python
def all_equal(iterable):
    """Returns True if all the elements are equal to each other""
    g = groupby(iterable)
    return next(g, True) and not next(g, False)
```

```python
def quantify(iterable, pred=bool):
    """Count how many times the predicate is true""
    return sum(map(pred, iterable))
```

```python
def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.
    Useful for emulating the behavior of the built-in map() function."
    return chain(iterable, repeat(None))
```

```python
def ncycles(iterable, n):
    """Returns the sequence elements n times""
    return chain.from_iterable(repeat(tuple(iterable), n))
```

```python
def dotproduct(vec1, vec2):
    """Flatten one level of nesting""
```

```python
def flatten(listOfLists):
    """Flatten one level of nesting""
```
```python
return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.
    Example: repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))
```
def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries
    # partition(is_odd, range(10)) --> 0 2 4 6 8 and 1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s) + 1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable))):

def iter_except(func, exception, first=None):
    """Call a function repeatedly until an exception is raised."
    try:
        while True:
            yield func()
Converts a call-until-exception interface to an iterator interface. Like `builtins.iter(func, sentinel)` but uses an exception instead of a sentinel to end the loop.

Examples:
```python
iter_except(functools.partial(heappop, h), IndexError)  # priority queue iterator
iter_except(d.popitem, KeyError)  # non-blocking dict iterator
iter_except(d.popleft, IndexError)  # non-blocking deque iterator
iter_except(q.get_nowait, Queue.Empty)  # loop over a producer Queue
iter_except(s.pop, KeyError)  # non-blocking set iterator
```

```python
try:
    if first is not None:
        yield first()  # For database
    while True:
        yield func()
except exception:
    pass
```

```python
def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*.

    If *pred* is not None, returns the first item for which pred(item) is true.
    
    """
    return next(filter(pred, iterable), default)
```

```python
def random_product(*args, repeat=1):
    """Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)
```
def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]' 
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
c, n, r = c*r//n, n-1, r-1
while index >= c:
    index -= c
    c, n = c*(n-r)//n, n-1
result.append(pool[-1-n])
return tuple(result)

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the *dotproduct* recipe can be written as:

```python
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
functools — Higher-order functions and operations on callable objects

Source code: Lib/functools.py

The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The functools module defines the following functions:

functools.cmp_to_key(func)
   Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as sorted(), min(), max(), heapq.nlargest(), heapq.nsmallest(), itertools.groupby()). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

   A comparison function is any callable that accept two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Example:

```python
sorted(iterable, key=cmp_to_key(locale.strcoll))
```
For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

New in version 3.2.

@functools.lru_cache(maxsize=128, typed=False)

Decorator to wrap a function with a memoizing callable that saves up to the maxsize most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, \( f(a=1, b=2) \) and \( f(b=2, a=1) \) differ in their keyword argument order and may have two separate cache entries.

If maxsize is set to None, the LRU feature is disabled and the cache can grow without bound. The LRU feature performs best when maxsize is a power-of-two.

If typed is set to true, function arguments of different types will be cached separately. For example, \( f(3) \) and \( f(3.0) \) will be treated as distinct calls with distinct results.

To help measure the effectiveness of the cache and tune the maxsize parameter, the wrapped function is instrumented with a cache_info() function that returns a named tuple showing hits, misses, maxsize and currsizes. In a multi-threaded environment, the hits and misses are approximate.
The decorator also provides a cache_clear() function for clearing or invalidating the cache.

The original underlying function is accessible through the __wrapped__ attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call, or impure functions such as time() or random().

Example of an LRU cache for static web content:

```python
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, ...
...    pep = get_pep(n)
...    print(n, len(pep))
```
Example of efficiently computing **Fibonacci numbers** using a cache to implement a **dynamic programming** technique:

```python
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

```python
>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

```python
>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=0)
```

New in version 3.2.

*Changed in version 3.3:* Added the **typed** option.

[@functools.total_ordering](https://docs.python.org/3/library/functools.html#functools.total_ordering)

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

For example:
@total_ordering
class Student:
    def __is_valid_operand__(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self.__is_valid_operand__(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname
                 (other.lastname.lower(), other.firstname
    def __lt__(self, other):
        if not self.__is_valid_operand__(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname
                 (other.lastname.lower(), other.firstname

Note: While this decorator makes it easy to create well behaved totally ordered types, it does come at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead is likely to provide an easy speed boost.

New in version 3.2.

Changed in version 3.4: Returning NotImplemented from the underlying comparison function for unrecognised types is now supported.

functools.partial(func, *args, **keywords)

Return a new partial object which when called will behave like func called with the positional arguments args and keyword arguments keywords. If more arguments are supplied to the call, they are
appended to *args. If additional keyword arguments are supplied, they extend and override *keywords. Roughly equivalent to:

```python
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The `partial()` is used for partial function application which “freezes” some portion of a function’s arguments and/or keywords resulting in a new object with a simplified signature. For example, `partial()` can be used to create a callable that behaves like the `int()` function where the `base` argument defaults to two:

```python
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

```python
class functools.partialmethod(func, *args, **keywords)
Return a new partialmethod descriptor which behaves like partial except that it is designed to be used as a method definition rather than being directly callable.

*func* must be a descriptor or a callable (objects which are both, like normal functions, are handled as descriptors).
```
When `func` is a descriptor (such as a normal Python function, `classmethod()`, `staticmethod()`, `abstractmethod()` or another instance of `partialmethod`), calls to `__get__` are delegated to the underlying descriptor, and an appropriate `partial object` returned as the result.

When `func` is a non-descriptor callable, an appropriate bound method is created dynamically. This behaves like a normal Python function when used as a method: the `self` argument will be inserted as the first positional argument, even before the `args` and `keywords` supplied to the `partialmethod` constructor.

Example:

```python
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)

>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

*New in version 3.4.*
functools.reduce(function, iterable[, initializer])

Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates (((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the sequence. If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

Roughly equivalent to:

```python
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

@functools.singledispatch

Transform a function into a single-dispatch generic function.

To define a generic function, decorate it with the @singledispatch decorator. Note that the dispatch happens on the type of the first argument, create your function accordingly:

```python
>>> from functools import singledispatch
>>> @singledispatch
```
... def fun(arg, verbose=False):
    if verbose:
        print("Let me just say,", end=" ")
    print(arg)

To add overloaded implementations to the function, use the `register()` attribute of the generic function. It is a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```python
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=
...     print(arg)
... 
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

For code which doesn't use type annotations, the appropriate type argument can be passed explicitly to the decorator itself:

```python
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=
...     print(arg.real, arg.imag)
```

To enable registering lambdas and pre-existing functions, the
register() attribute can be used in a functional form:

```python
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
... >>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function which enables decorator stacking, pickling, as well as creating unit tests for each variant independently:

```python
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...         print(arg / 2)
... >>> fun_num is fun
False
```

When called, the generic function dispatches on the type of the first argument:

```python
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(["spam", "spam", "eggs", "spam"], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
```
Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with \texttt{@singledispatch} is registered for the base \texttt{object} type, which means it is used if no better implementation is found.

To check which implementation will the generic function choose for a given type, use the \texttt{dispatch()} attribute:

```python
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only \texttt{registry} attribute:

```python
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>, <class 'decimal.Decimal'>, <class 'list'>, <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

\textit{New in version 3.4.}
Changed in version 3.7: The register() attribute supports using type annotations.

```python
functools.update_wrapper(wrapper, wrapped,
assigned=WRAPPER_ASSIGNMENTS,
updated=WRAPPER_UPDATES)
```

Update a `wrapper` function to look like the `wrapped` function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's `__module__`, `__name__`, `__qualname__`, `__annotations__` and `__doc__`, the documentation string) and `WRAPPER_UPDATES` (which updates the wrapper function's `__dict__`, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as `lru_cache()`), this function automatically adds a `__wrapped__` attribute to the wrapper that refers to the function being wrapped.

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`update_wrapper()` may be used with callables other than functions. Any attributes named in `assigned` or `updated` that are missing from the object being wrapped are ignored (i.e. this
function will not attempt to set them on the wrapper function).  
\texttt{AttributeError} is still raised if the wrapper function itself is 
missing any attributes named in \texttt{updated}.

\textit{New in version 3.2:} Automatic addition of the \texttt{\_\_wrapped\_} 
attribute.

\textit{New in version 3.2:} Copying of the \texttt{\_\_annotations\_} attribute 
by default.

\textit{Changed in version 3.2:} Missing attributes no longer trigger an 
\texttt{AttributeError}.

\textit{Changed in version 3.4:} The \texttt{\_\_wrapped\_} attribute now always 
refers to the wrapped function, even if that function defined a 
\texttt{\_\_wrapped\_} attribute. (see \texttt{bpo-17482})

\begin{verbatim}
@functools.wraps(wrapped, 
assigned=WRAPPER_ASSIGNMENTS, 
updated=WRAPPER_UPDATES)

This is a convenience function for invoking \texttt{update_wrapper()} 
as a function decorator when defining a wrapper function. It is 
equivalent to \texttt{partial(update_wrapper, 
wrapped=wrapped, 
assigned=assigned, 
updated=updated)}. For example:

```python
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
```
Without the use of this decorator factory, the name of the example function would have been `wrapper`, and the docstring of the original `example()` would have been lost.
partial Objects

partial objects are callable objects created by partial(). They have three read-only attributes:

partial.func
A callable object or function. Calls to the partial object will be forwarded to func with new arguments and keywords.

partial.args
The leftmost positional arguments that will be prepended to the positional arguments provided to a partial object call.

partial.keywords
The keyword arguments that will be supplied when the partial object is called.

partial objects are like function objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the __name__ and __doc__ attributes are not created automatically. Also, partial objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.
The `operator` module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. Many function names are those used for special methods, without the double underscores. For backward compatibility, many of these have a variant with the double underscores kept. The variants without the double underscores are preferred for clarity.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

```python
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
```
operator.__gt__(a, b)
Perform “rich comparisons” between $a$ and $b$. Specifically, $\lt(a, b)$ is equivalent to $a < b$, $\leq(a, b)$ is equivalent to $a \leq b$, $\text{eq}(a, b)$ is equivalent to $a == b$, $\text{ne}(a, b)$ is equivalent to $a \neq b$, $\text{gt}(a, b)$ is equivalent to $a > b$ and $\text{ge}(a, b)$ is equivalent to $a \geq b$. Note that these functions can return any value, which may or may not be interpretable as a Boolean value. See \texttt{Comparisons} for more information about rich comparisons.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

operator.\texttt{not}_(obj)
operator.\texttt{__not__}(obj)
Return the outcome of \texttt{not} obj. (Note that there is no \texttt{__not__}() method for object instances; only the interpreter core defines this operation. The result is affected by the \texttt{__bool__}() and \texttt{__len__}() methods.)

operator.\texttt{truth}(obj)
Return \texttt{True} if obj is true, and \texttt{False} otherwise. This is equivalent to using the \texttt{bool} constructor.

operator.\texttt{is}_(a, b)
Return $a$ is $b$. Tests object identity.

operator.\texttt{is\_not}_(a, b)
Return $a$ is not $b$. Tests object identity.

The mathematical and bitwise operations are the most numerous:

operator.\texttt{abs}(obj)
operator.__abs__(obj)
    Return the absolute value of obj.

operator.add(a, b)
operator.__add__(a, b)
    Return a + b, for a and b numbers.

operator.and_(a, b)
operator.__and__(a, b)
    Return the bitwise and of a and b.

operator.floordiv(a, b)
operator.__floordiv__(a, b)
    Return a // b.

operator.index(a)
operator.__index__(a)
    Return a converted to an integer. Equivalent to a.__index__().

operator.inv(obj)
operator.invert(obj)
operator.__inv__(obj)
operator.__invert__(obj)
    Return the bitwise inverse of the number obj. This is equivalent to ~obj.

operator.lshift(a, b)
operator.__lshift__(a, b)
    Return a shifted left by b.

operator.mod(a, b)
operator.__mod__(a, b)
Return \( a \% b \).

\[
\text{operator.} \text{\texttt{mul}}(a, b) \\
\text{operator.} \text{\texttt{\_\_mul\_\_}}(a, b)
\]

Return \( a \ast b \), for \( a \) and \( b \) numbers.

\[
\text{operator.} \text{\texttt{matmul}}(a, b) \\
\text{operator.} \text{\texttt{\_\_matmul\_\_}}(a, b)
\]

Return \( a \ast b \).

*New in version 3.5.*

\[
\text{operator.} \text{\texttt{neg}}(obj) \\
\text{operator.} \text{\texttt{\_\_neg\_\_}}(obj)
\]

Return \( obj \) negated (-\( obj \)).

\[
\text{operator.} \text{\texttt{or\_\_}}(a, b) \\
\text{operator.} \text{\texttt{\_\_or\_\_}}(a, b)
\]

Return the bitwise or of \( a \) and \( b \).

\[
\text{operator.} \text{\texttt{pos}}(obj) \\
\text{operator.} \text{\texttt{\_\_pos\_\_}}(obj)
\]

Return \( obj \) positive (+\( obj \)).

\[
\text{operator.} \text{\texttt{pow}}(a, b) \\
\text{operator.} \text{\texttt{\_\_pow\_\_}}(a, b)
\]

Return \( a \ast b \), for \( a \) and \( b \) numbers.

\[
\text{operator.} \text{\texttt{rshift}}(a, b) \\
\text{operator.} \text{\texttt{\_\_rshift\_\_}}(a, b)
\]

Return \( a \) shifted right by \( b \).

\[
\text{operator.} \text{\texttt{sub}}(a, b)
\]
operator.__sub__(a, b)

Return a - b.

operator.truediv(a, b)
operator.__truediv__(a, b)

Return a / b where 2/3 is .66 rather than 0. This is also known as “true” division.

operator.xor(a, b)
operator.__xor__(a, b)

Return the bitwise exclusive or of a and b.

Operations which work with sequences (some of them with mappings too) include:

operator.concat(a, b)
operator.__concat__(a, b)

Return a + b for a and b sequences.

operator.contains(a, b)
operator.__contains__(a, b)

Return the outcome of the test b in a. Note the reversed operands.

operator.countOf(a, b)

Return the number of occurrences of b in a.

operator.delitem(a, b)
operator.__delitem__(a, b)

Remove the value of a at index b.

operator.getitem(a, b)
operator.__getitem__(a, b)
Return the value of $a$ at index $b$.

\texttt{operator.indexof}(a, b)

Return the index of the first of occurrence of $b$ in $a$.

\texttt{operator.setitem}(a, b, c)
\texttt{operator.__setitem__}(a, b, c)

Set the value of $a$ at index $b$ to $c$.

\texttt{operator.length_hint}(obj, default=0)

Return an estimated length for the object $o$. First try to return its actual length, then an estimate using \texttt{object.__length_hint__()}, and finally return the default value.

\textit{New in version 3.4.}\n
The \texttt{operator} module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for \texttt{map()}, \texttt{sorted()}, \texttt{itertools.groupby()}, or other functions that expect a function argument.

\texttt{operator.attrgetter}(\texttt{attr})
\texttt{operator.attrgetter}(\*\texttt{attrs})

Return a callable object that fetches $attr$ from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After $f = \texttt{attrgetter('name')}$, the call $f(b)$ returns $b.name$.
- After $f = \texttt{attrgetter('name', 'date')}$, the call $f(b)$ returns ($b.name$, $b.date$).
- After $f = \texttt{attrgetter('name.first', 'name.last')}$,
the call \( f(b) \) returns \((b\text{.name.first, } b\text{.name.last})\).

Equivalent to:

```python
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for return g

def resolve_attr(obj, attr):
    for name in attr.split('.'):  
        obj = getattr(obj, name)
    return obj
```

operator.\texttt{itemgetter(item)}
operator.\texttt{itemgetter(*items)}

Return a callable object that fetches \texttt{item} from its operand using the operand's \texttt{__getitem__()} method. If multiple items are specified, returns a tuple of lookup values. For example:

- After \( f = \texttt{itemgetter(2)} \), the call \( f(r) \) returns \( r[2] \).
- After \( g = \texttt{itemgetter(2, 5, 3)} \), the call \( g(r) \) returns \( (r[2], r[5], r[3]) \).
item = items[0]
def g(obj):
    return obj[item]
else:
    def g(obj):
        return tuple(obj[item] for item in it)
return g

The items can be any type accepted by the operand's `__getitem__()` method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

```python
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'

>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Example of using `itemgetter()` to retrieve specific fields from a tuple record:

```python
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```
operator. **methodcaller**(name[, args...])

Return a callable object that calls the method name on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.
- After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1).

Equivalent to:

```python
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    returncaller
```
Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td><code>a + b</code></td>
<td><code>add(a, b)</code></td>
</tr>
<tr>
<td>Concatenation</td>
<td><code>seq1 + seq2</code></td>
<td><code>concat(seq1, seq2)</code></td>
</tr>
<tr>
<td>Containment Test</td>
<td><code>obj in seq</code></td>
<td><code>contains(seq, obj)</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
<td><code>truediv(a, b)</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a // b</code></td>
<td><code>floordiv(a, b)</code></td>
</tr>
<tr>
<td>Bitwise And</td>
<td><code>a &amp; b</code></td>
<td><code>and_(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Exclusive Or</td>
<td><code>a ^ b</code></td>
<td><code>xor(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Inversion</td>
<td><code>~ a</code></td>
<td><code>invert(a)</code></td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td>Exponentiation</td>
<td><code>a ** b</code></td>
<td><code>pow(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is b</code></td>
<td><code>is_(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is not b</code></td>
<td><code>is_not(a, b)</code></td>
</tr>
<tr>
<td>Indexed Assignment</td>
<td><code>obj[k] = v</code></td>
<td><code>setitem(obj, k, v)</code></td>
</tr>
<tr>
<td>Indexed Deletion</td>
<td><code>del obj[k]</code></td>
<td><code>delitem(obj, k)</code></td>
</tr>
<tr>
<td>Indexing</td>
<td><code>obj[k]</code></td>
<td><code>getitem(obj, k)</code></td>
</tr>
<tr>
<td>Operation</td>
<td>Symbol</td>
<td>Function</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>Left Shift</td>
<td>a &lt;&lt; b</td>
<td>lshift(a, b)</td>
</tr>
<tr>
<td>Modulo</td>
<td>a % b</td>
<td>mod(a, b)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>a * b</td>
<td>mul(a, b)</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>a @ b</td>
<td>matmul(a, b)</td>
</tr>
<tr>
<td>Negation (Arithmetic)</td>
<td>- a</td>
<td>neg(a)</td>
</tr>
<tr>
<td>Negation (Logical)</td>
<td>not a</td>
<td>not_(a)</td>
</tr>
<tr>
<td>Positive</td>
<td>+ a</td>
<td>pos(a)</td>
</tr>
<tr>
<td>Right Shift</td>
<td>a &gt;&gt; b</td>
<td>rshift(a, b)</td>
</tr>
<tr>
<td>Slice Assignment</td>
<td>seq[i:j] = values</td>
<td>setitem(seq, slice(i, j), values)</td>
</tr>
<tr>
<td>Slice Deletion</td>
<td>del seq[i:j]</td>
<td>delitem(seq, slice(i, j))</td>
</tr>
<tr>
<td>Slicing</td>
<td>seq[i:j]</td>
<td>getitem(seq, slice(i, j))</td>
</tr>
<tr>
<td>String Formatting</td>
<td>s % obj</td>
<td>mod(s, obj)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>a - b</td>
<td>sub(a, b)</td>
</tr>
<tr>
<td>Truth Test</td>
<td>obj</td>
<td>truth(obj)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &lt; b</td>
<td>lt(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &lt;= b</td>
<td>le(a, b)</td>
</tr>
<tr>
<td>Equality</td>
<td>a == b</td>
<td>eq(a, b)</td>
</tr>
<tr>
<td>Difference</td>
<td>a != b</td>
<td>ne(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &gt;= b</td>
<td>ge(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &gt; b</td>
<td>gt(a, b)</td>
</tr>
</tbody>
</table>
Inplace Operators

Many operations have an “in-place” version. Listed below are functions providing a more primitive access to in-place operators than the usual syntax does; for example, the statement `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

In those examples, note that when an in-place method is called, the computation and assignment are performed in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back to the input variable:

```python
>>> a = 'hello'
>>> iadd(a, ' world')
hello world
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the inplace method will perform the update, so no subsequent assignment is necessary:

```python
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```
operator. iadd(a, b)
operator. __iadd__(a, b)
    a = iadd(a, b) is equivalent to a += b.

operator. iand(a, b)
operator. __iand__(a, b)
    a = iand(a, b) is equivalent to a &= b.

operator. iconcat(a, b)
operator. __iconcat__(a, b)
    a = iconcat(a, b) is equivalent to a += b for a and b sequences.

operator. ifloordiv(a, b)
operator. __ifloordiv__(a, b)
    a = ifloordiv(a, b) is equivalent to a //= b.

operator. ilshift(a, b)
operator. __ilshift__(a, b)
    a = ilshift(a, b) is equivalent to a <<= b.

operator. imod(a, b)
operator. __imod__(a, b)
    a = imod(a, b) is equivalent to a %= b.

operator. imul(a, b)
operator. __imul__(a, b)
    a = imul(a, b) is equivalent to a *= b.

operator. imatmul(a, b)
operator. __imatmul__(a, b)
    a = imatmul(a, b) is equivalent to a @= b.
New in version 3.5.

operator.\texttt{\_\_ior\_}\,(a,\,b)
operator.\texttt{\_\_ior\_\_}\,(a,\,b)
\begin{equation}
a = \text{ior}(a,\,b) \text{ is equivalent to } a | = b.
\end{equation}

operator.\texttt{\_ipow\_}\,(a,\,b)
operator.\texttt{\_\_ipow\_\_}\,(a,\,b)
\begin{equation}
a = \text{ipow}(a,\,b) \text{ is equivalent to } a **= b.
\end{equation}

operator.\texttt{\_irshift\_}\,(a,\,b)
operator.\texttt{\_\_irshift\_\_}\,(a,\,b)
\begin{equation}
a = \text{irshift}(a,\,b) \text{ is equivalent to } a >>= b.
\end{equation}

operator.\texttt{\_isub\_}\,(a,\,b)
operator.\texttt{\_\_isub\_\_}\,(a,\,b)
\begin{equation}
a = \text{isub}(a,\,b) \text{ is equivalent to } a -= b.
\end{equation}

operator.\texttt{\_itruediv\_}\,(a,\,b)
operator.\texttt{\_\_itruediv\_\_}\,(a,\,b)
\begin{equation}
a = \text{itruediv}(a,\,b) \text{ is equivalent to } a /= b.
\end{equation}

operator.\texttt{\_ixor\_}\,(a,\,b)
operator.\texttt{\_\_ixor\_\_}\,(a,\,b)
\begin{equation}
a = \text{ixor}(a,\,b) \text{ is equivalent to } a ^= b.
\end{equation}
The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

- **pathlib** — Object-oriented filesystem paths
  - Basic use
  - Pure paths
    - General properties
    - Operators
    - Accessing individual parts
    - Methods and properties
  - Concrete paths
    - Methods
  - Correspondence to tools in the os module
- **os.path** — Common pathname manipulations
- **fileinput** — Iterate over lines from multiple input streams
- **stat** — Interpreting stat() results
- **filecmp** — File and Directory Comparisons
  - The dircmp class
- **tempfile** — Generate temporary files and directories
  - Examples
  - Deprecated functions and variables
- **glob** — Unix style pathname pattern expansion
- **fnmatch** — Unix filename pattern matching
- **linecache** — Random access to text lines
- **shutil** — High-level file operations
  - Directory and files operations
    - copytree example
    - rmtree example
- Archiving operations
  - Archiving example
- Querying the size of the output terminal
- `macpath` — Mac OS 9 path manipulation functions

**See also:**

**Module** `os`
Operating system interfaces, including functions to work with files at a lower level than Python `file objects`.

**Module** `io`
Python’s built-in I/O library, including both abstract classes and some concrete classes such as file I/O.

**Built-in function** `open()`
The standard way to open files for reading and writing with Python.
**pathlib** — Object-oriented filesystem paths

*New in version 3.4.*

**Source code:** Lib/pathlib.py

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.

If you’ve never used this module before or just aren’t sure which class is right for your task, **Path** is most likely what you need. It instantiates a concrete path for the platform the code is running on.

Pure paths are useful in some special cases; for example:
1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a `WindowsPath` when running on Unix, but you can instantiate `PureWindowsPath`.

2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

**See also:** [PEP 428](https://www.python.org/dev/peps/pep-0428/): The `pathlib` module – object-oriented filesystem paths.

**See also:** For low-level path manipulation on strings, you can also use the `os.path` module.
Basic use

Importing the main class:

```python
>>> from pathlib import Path
```

Listing subdirectories:

```python
>>> p = Path('.

>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```python
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
PosixPath('build/lib/pathlib.py')]
```

Navigating inside a directory tree:

```python
>>> p = Path('/etc')

>>> q = p / 'init.d' / 'reboot'

>>> q
PosixPath('/etc/init.d/reboot')

>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Querying path properties:

```python
>>> q.exists()
True
```
>>> q.is_dir()
False

Opening a file:

```python
>>> with q.open() as f: f.readline()
... '#!/bin/bash
'
```
Pure paths

Pure path objects provide path-handling operations which don't actually access a filesystem. There are three ways to access these classes, which we also call *flavours*:

```python
class pathlib.PurePath(*pathsegments)
```

A generic class that represents the system's path flavour (instantiating it creates either a `PurePosixPath` or a `PureWindowsPath`):

```python
>>> PurePath('setup.py')  # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of `pathsegments` can be either a string representing a path segment, an object implementing the `os.PathLike` interface which returns a string, or another path object:

```python
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

When `pathsegments` is empty, the current directory is assumed:

```python
>>> PurePath()
PurePosixPath('.')
```

When several absolute paths are given, the last is taken as an anchor (mimicking `os.path.join()`'s behaviour):

```python
>>> PurePath('/etc', '/usr', 'lib64')
```
However, in a Windows path, changing the local root doesn't discard the previous drive setting:

```python
given PureWindowsPath('c:/Windows', 'd:bar')
```

Spurious slashes and single dots are collapsed, but double dots ('..') are not, since this would change the meaning of a path in the face of symbolic links:

```python
given PurePath('foo//bar')
PurePosixPath('foo/bar')
given PurePath('foo/./bar')
PurePosixPath('foo/bar')
given PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(a naïve approach would make
PurePosixPath('foo/../bar') equivalent to
PurePosixPath('bar'), which is wrong if foo is a symbolic link to another directory)

Pure path objects implement the `os.PathLike` interface, allowing them to be used anywhere the interface is accepted.

*Changed in version 3.6:* Added support for the `os.PathLike` interface.

```python
class pathlib.PurePosixPath(*pathsegments)
```
A subclass of `PurePath`, this path flavour represents non-Windows filesystem paths:

```python
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

`pathsegments` is specified similarly to `PurePath`.

class `pathlib.PureWindowsPath(*pathsegments)`

A subclass of `PurePath`, this path flavour represents Windows filesystem paths:

```python
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

`pathsegments` is specified similarly to `PurePath`.

Regardless of the system you’re running on, you can instantiate all of these classes, since they don’t provide any operation that does system calls.

**General properties**

Paths are immutable and hashable. Paths of a same flavour are comparable and orderable. These properties respect the flavour’s case-folding semantics:

```python
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo'), PureWindowsPath('FOO') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
```
Paths of a different flavour compare unequal and cannot be ordered:

```python
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
```

Operators

The slash operator helps create child paths, similarly to `os.path.join()`:

```python
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

A path object can be used anywhere an object implementing `os.PathLike` is accepted:

```python
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```
The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```python
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\Program Files'
```

Similarly, calling `bytes` on a path gives the raw filesystem path as a bytes object, as encoded by `os.fsencode()`:

```python
>>> bytes(p)
b'/etc'
```

**Note:** Calling `bytes` is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

**Accessing individual parts**

To access the individual “parts” (components) of a path, use the following property:

`PurePath.parts`

A tuple giving access to the path's various components:

```python
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')
```
```python
>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\', 'Program Files', 'PSF')
```

(note how the drive and local root are regrouped in a single part)

Methods and properties

Pure paths provide the following methods and properties:

**PurePath**. **drive**

A string representing the drive letter or name, if any:

```python
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```python
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\host\share'
```

**PurePath**. **root**

A string representing the (local or global) root, if any:

```python
>>> PureWindowsPath('c:/Program Files/').root
'\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```
UNC shares always have a root:

```python
>>> PureWindowsPath('//host/share').root
'\\'
```

**PurePath. anchor**

The concatenation of the drive and root:

```python
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\host\\share\\'
```

**PurePath. parents**

An immutable sequence providing access to the logical ancestors of the path:

```python
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

**PurePath. parent**

The logical parent of the path:

```python
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```
You cannot go past an anchor, or empty path:

```python
code3
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

**Note:** This is a purely lexical operation, hence the following behaviour:

```python
code3
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

If you want to walk an arbitrary filesystem path upwards, it is recommended to first call `Path.resolve()` so as to resolve symlinks and eliminate “..” components.

**PurePath.** `name`

A string representing the final path component, excluding the drive and root, if any:

```python
code3
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC drive names are not considered:

```python
code3
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```
PurePath. **suffix**

The file extension of the final component, if any:

```python
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath. **suffixes**

A list of the path's file extensions:

```python
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath. **stem**

The final path component, without its suffix:

```python
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath. **as_posix()**

Return a string representation of the path with forward slashes (/):

```python
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
```
PurePath.\texttt{as\_uri()}

Represent the path as a \texttt{file} URI. \texttt{ValueError} is raised if the path isn't absolute.

```python
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.\texttt{is\_absolute()}

Return whether the path is absolute or not. A path is considered absolute if it has both a root and (if the flavour allows) a drive:

```python
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.\texttt{is\_reserved()}
With `PureWindowsPath`, return `True` if the path is considered reserved under Windows, `False` otherwise. With `PurePosixPath`, `False` is always returned.

```python
g>>> PureWindowsPath('nul').is_reserved()
True
g>>> PurePosixPath('nul').is_reserved()
False
```

File system calls on reserved paths can fail mysteriously or have unintended effects.

`PurePath`. `joinpath(*other)`

Calling this method is equivalent to combining the path with each of the `other` arguments in turn:

```python
g>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
g>>> PurePosixPath('/etc').joinpath(PurePosixPath('/etc/passwd'))
PurePosixPath('/etc/passwd')
g>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
g>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

`PurePath`. `match(pattern)`

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise.

If `pattern` is relative, the path can be either relative or absolute, and matching is done from the right:

```python
g>>> PurePath('a/b.py').match('*.py')
True
```
PurePath. `relative_to(*other)`

Compute a version of this path relative to the path represented by *other*. If it's impossible, `ValueError` is raised:

```python
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/u'
```
PurePath.\texttt{with\_name}(name)

Return a new path with the \texttt{name} changed. If the original path
doesn't have a name, ValueError is raised:

```python
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py"
    raise ValueError("%r has an empty name" % (self)
ValueError: PureWindowsPath('c:/') has an empty n
```

PurePath.\texttt{with\_suffix}(suffix)

Return a new path with the \texttt{suffix} changed. If the original path
doesn't have a suffix, the new \texttt{suffix} is appended instead. If the
\texttt{suffix} is an empty string, the original suffix is removed:

```python
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```
Concrete paths

Concrete paths are subclasses of the pure path classes. In addition to operations provided by the latter, they also provide methods to do system calls on path objects. There are three ways to instantiate concrete paths:

**class pathlib.Path(*pathsegments)**

A subclass of PurePath, this class represents concrete paths of the system’s path flavour (instantiating it creates either a PosixPath or a WindowsPath):

```python
>>> Path('setup.py')
PosixPath('setup.py')
```

*pathsegments* is specified similarly to PurePath.

**class pathlib.PosixPath(*pathsegments)**

A subclass of Path and PurePosixPath, this class represents concrete non-Windows filesystem paths:

```python
>>> PosixPath('/etc')
PosixPath('/etc')
```

*pathsegments* is specified similarly to PurePath.

**class pathlib.WindowsPath(*pathsegments)**

A subclass of Path and PureWindowsPath, this class represents concrete Windows filesystem paths:

```python
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```
pathsegments is specified similarly to PurePath.

You can only instantiate the class flavour that corresponds to your system (allowing system calls on non-compatible path flavours could lead to bugs or failures in your application):

```python
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath'
```

**Methods**

Concrete paths provide the following methods in addition to pure paths methods. Many of these methods can raise an OSError if a system call fails (for example because the path doesn't exist):

**classmethod** `Path.cwd()`

Return a new path object representing the current directory (as returned by `os.getcwd()`):

```python
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```
**classmethod** `Path.home()`

Return a new path object representing the user’s home directory (as returned by `os.path.expanduser()` with ~ construct):

```python
>>> Path.home()
PosixPath('/home/antoine')
```

*New in version 3.5.*

**Path.stat()**

Return information about this path (similarly to `os.stat()`). The result is looked up at each call to this method.

```python
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

**Path.chmod(mode)**

Change the file mode and permissions, like `os.chmod()`:

```python
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

**Path.exists()**

Whether the path points to an existing file or directory:

```python
>>> Path('.').exists()
True
```
```python
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

**Note:** If the path points to a symlink, `exists()` returns whether the symlink *points to* an existing file or directory.

**Path.** `expanduser()`

Return a new path with expanded ~ and ~user constructs, as returned by `os.path.expanduser()`:

```python
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

*New in version 3.5.*

**Path.** `glob(pattern)`

Glob the given relative *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```python
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'),
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

The "**" pattern means “this directory and all subdirectories, recursively”. In other words, it enables recursive globbing:

```python
>>> sorted(Path('.').glob('**/*.py'))
```
Note: Using the "**" pattern in large directory trees may consume an inordinate amount of time.

Path. **group()**
Return the name of the group owning the file. **KeyError** is raised if the file's gid isn't found in the system database.

Path. **is_dir()**
Return **True** if the path points to a directory (or a symbolic link pointing to a directory), **False** if it points to another kind of file.

**False** is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

Path. **is_file()**
Return **True** if the path points to a regular file (or a symbolic link pointing to a regular file), **False** if it points to another kind of file.

**False** is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

Path. **is_mount()**
Return **True** if the path is a mount point: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether path’s parent, path/..., is on a different device than path, or whether path/... and path point to the same
i-node on the same device — this should detect mount points for all Unix and POSIX variants. Not implemented on Windows.

*New in version 3.7.*

**Path**. *is_symlink()*

Return *True* if the path points to a symbolic link, *False* otherwise.

*False* is also returned if the path doesn’t exist; other errors (such as permission errors) are propagated.

**Path**. *is_socket()*

Return *True* if the path points to a Unix socket (or a symbolic link pointing to a Unix socket), *False* if it points to another kind of file.

*False* is also returned if the path doesn’t exist or is a broken symlink; other errors (such as permission errors) are propagated.

**Path**. *is_fifo()*

Return *True* if the path points to a FIFO (or a symbolic link pointing to a FIFO), *False* if it points to another kind of file.

*False* is also returned if the path doesn’t exist or is a broken symlink; other errors (such as permission errors) are propagated.

**Path**. *is_block_device()*

Return *True* if the path points to a block device (or a symbolic link pointing to a block device), *False* if it points to another kind of file.

*False* is also returned if the path doesn’t exist or is a broken symlink; other errors (such as permission errors) are propagated.

**Path**. *is_char_device()*
Return `True` if the path points to a character device (or a symbolic link pointing to a character device), `False` if it points to another kind of file.

`False` is also returned if the path doesn’t exist or is a broken symlink; other errors (such as permission errors) are propagated.

**Path. `iterdir()`**

When the path points to a directory, yield path objects of the directory contents:

```python
>>> p = Path('docs')
>>> for child in p.iterdir():
...     child
...     PosixPath('docs/conf.py')
     PosixPath('docs/_templates')
     PosixPath('docs/make.bat')
     PosixPath('docs/index.rst')
     PosixPath('docs/_build')
     PosixPath('docs/_static')
     PosixPath('docs/Makefile')
```

**Path. `lchmod(mode)`**

Like `Path.chmod()` but, if the path points to a symbolic link, the symbolic link’s mode is changed rather than its target’s.

**Path. `lstat()`**

Like `Path.stat()` but, if the path points to a symbolic link, return the symbolic link’s information rather than its target’s.

**Path. `mkdir(mode=0o777, parents=False, exist_ok=False)`**

Create a new directory at this given path. If `mode` is given, it is combined with the process’ umask value to determine the file mode and access flags. If the path already exists,
FileExistsError is raised.

If parents is true, any missing parents of this path are created as needed; they are created with the default permissions without taking mode into account (mimicking the POSIX mkdir -p command).

If parents is false (the default), a missing parent raises FileNotFoundError.

If exist_ok is false (the default), FileExistsError is raised if the target directory already exists.

If exist_ok is true, FileExistsError exceptions will be ignored (same behavior as the POSIX mkdir -p command), but only if the last path component is not an existing non-directory file.

Changed in version 3.5: The exist_ok parameter was added.

Path.open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)

Open the file pointed to by the path, like the built-in open() function does:

```python
>>> p = Path('setup.py')
>>> with p.open() as f:
...   f.readline()
...
'#!/usr/bin/env python3
'
```

Path.owner()

Return the name of the user owning the file. KeyError is raised if the file's uid isn’t found in the system database.
**Path. `read_bytes()`**

Return the binary contents of the pointed-to file as a bytes object:

```python
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

*New in version 3.5.*

**Path. `read_text()`**

```python
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in `open()`.

*New in version 3.5.*

**Path. `rename(target)`**

Rename this file or directory to the given `target`. On Unix, if `target` exists and is a file, it will be replaced silently if the user has permission. `target` can be either a string or another path object:

```python
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
```
```python
>>> target.open().read()
'some text'
```

Path. **replace**(target)

Rename this file or directory to the given target. If target points to an existing file or directory, it will be unconditionally replaced.

Path. **resolve**(strict=False)

Make the path absolute, resolving any symlinks. A new path object is returned:

```python
>>> p = Path()
>>> p
PosixPath('.
')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

“..” components are also eliminated (this is the only method to do so):

```python
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If the path doesn't exist and strict is True, FileNotFoundError is raised. If strict is False, the path is resolved as far as possible and any remainder is appended without checking whether it exists. If an infinite loop is encountered along the resolution path, RuntimeError is raised.

*New in version 3.6: The strict argument.*

Path. **rglob**(pattern)

This is like calling Path.glob() with ``/**`` added in front of the
given relative pattern:

```python
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

**Path.rmdir()**

Remove this directory. The directory must be empty.

**Path.samefile(other_path)**

Return whether this path points to the same file as `other_path`, which can be either a Path object, or a string. The semantics are similar to `os.path.samefile()` and `os.path.samestat()`.

An `OSError` can be raised if either file cannot be accessed for some reason.

```python
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

*New in version 3.5.*

**Path.symlink_to(target, target_is_directory=False)**

Make this path a symbolic link to `target`. Under Windows, `target_is_directory` must be true (default `False`) if the link’s target is a directory. Under POSIX, `target_is_directory`’s value is ignored.
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8

**Note:** The order of arguments (link, target) is the reverse of `os.symlink()`'s.

**Path.** `touch` *mode=* `0o666`, `exist_ok=True`*

Create a file at this given path. If *mode* is given, it is combined with the process' umask value to determine the file mode and access flags. If the file already exists, the function succeeds if `exist_ok` is true (and its modification time is updated to the current time), otherwise *FileExistsError* is raised.

**Path.** `unlink()`

Remove this file or symbolic link. If the path points to a directory, use Path.**rmdir()** instead.

**Path.** `write_bytes(data)`

Open the file pointed to in bytes mode, write *data* to it, and close the file:

```python
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
```

b'Binary file contents'
An existing file of the same name is overwritten.

New in version 3.5.

Path.

Path

write_text(data, encoding=None, errors=None)

Open the file pointed to in text mode, write data to it, and close the file:

```py
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

New in version 3.5.
Correspondence to tools in the `os` module

Below is a table mapping various `os` functions to their corresponding `PurePath/Path` equivalent.

**Note:** Although `os.path.relpath()` and `PurePath.relative_to()` have some overlapping use-cases, their semantics differ enough to warrant not considering them equivalent.

<table>
<thead>
<tr>
<th><code>os and os.path</code></th>
<th><code>pathlib</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os.path.abspath()</code></td>
<td><code>Path.resolve()</code></td>
</tr>
<tr>
<td><code>os.chmod()</code></td>
<td><code>Path.chmod()</code></td>
</tr>
<tr>
<td><code>os.mkdir()</code></td>
<td><code>Path.mkdir()</code></td>
</tr>
<tr>
<td><code>os.rename()</code></td>
<td><code>Path.rename()</code></td>
</tr>
<tr>
<td><code>os.replace()</code></td>
<td><code>Path.replace()</code></td>
</tr>
<tr>
<td><code>os.rmdir()</code></td>
<td><code>Path.rmdir()</code></td>
</tr>
<tr>
<td><code>os.remove(), os.unlink()</code></td>
<td><code>Path.unlink()</code></td>
</tr>
<tr>
<td><code>os.getcwd()</code></td>
<td><code>Path.cwd()</code></td>
</tr>
<tr>
<td><code>os.path.exists()</code></td>
<td><code>Path.exists()</code></td>
</tr>
<tr>
<td><code>os.path.expanduser()</code></td>
<td><code>Path.expanduser()</code> and <code>Path.home()</code></td>
</tr>
<tr>
<td><code>os.path.isdir()</code></td>
<td><code>Path.is_dir()</code></td>
</tr>
<tr>
<td><code>os.path.isfile()</code></td>
<td><code>Path.is_file()</code></td>
</tr>
<tr>
<td><code>os.path.islink()</code></td>
<td><code>Path.is_symlink()</code></td>
</tr>
<tr>
<td><code>os.stat()</code></td>
<td><code>Path.stat()</code>, <code>Path.owner()</code></td>
</tr>
<tr>
<td>Function</td>
<td>Method</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>os.path.isabs()</td>
<td>PurePath.is_absolute()</td>
</tr>
<tr>
<td>os.path.join()</td>
<td>PurePath.joinpath()</td>
</tr>
<tr>
<td>os.path.basename()</td>
<td>PurePath.name</td>
</tr>
<tr>
<td>os.path.dirname()</td>
<td>PurePath.parent</td>
</tr>
<tr>
<td>os.path.samefile()</td>
<td>Path.samefile()</td>
</tr>
<tr>
<td>os.path.splitext()</td>
<td>PurePath.suffix</td>
</tr>
</tbody>
</table>

Path.group()
os.path — Common pathname manipulations

Source code: Lib/posixpath.py (for POSIX), Lib/ntpath.py (for Windows NT), and Lib/macpath.py (for Macintosh)

This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings. Unfortunately, some file names may not be representable as strings on Unix, so applications that need to support arbitrary file names on Unix should use bytes objects to represent path names. Vice versa, using bytes objects cannot represent all file names on Windows (in the standard mbcs encoding), hence Windows applications should use string objects to access all files.

Unlike a unix shell, Python does not do any automatic path expansions. Functions such as expanduser() and expandvars() can be invoked explicitly when an application desires shell-like path expansion. (See also the glob module.)

See also: The pathlib module offers high-level path objects.

Note: All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.
Note: Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The `os.path` module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is always in one of the different formats. They all have the same interface:

- `posixpath` for UNIX-style paths
- `ntpath` for Windows paths
- `macpath` for old-style MacOS paths

`os.path.abspath(path)`

Return a normalized absolutized version of the pathname `path`. On most platforms, this is equivalent to calling the function `normpath()` as follows: `normpath(join(os.getcwd(), path))`.

*Changed in version 3.6:* Accepts a path-like object.

`os.path.basename(path)`

Return the base name of pathname `path`. This is the second element of the pair returned by passing `path` to the function `split()`. Note that the result of this function is different from the Unix `basename` program; where `basename` for `'/foo/bar/'` returns `'bar'`, the `basename()` function returns an empty string (`''`).

*Changed in version 3.6:* Accepts a path-like object.

`os.path.commonpath(paths)`

Return the longest common sub-path of each pathname in the
sequence *paths*. Raise `ValueError` if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike `commonprefix()`, this returns a valid path.

**Availability:** Unix, Windows.

*New in version 3.5.*

*Changed in version 3.6:* Accepts a sequence of *path-like objects*.

```python
os.path.commonprefix(list)
```

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string ("").

**Note:** This function may return invalid paths because it works a character at a time. To obtain a valid path, see `commonpath()`.

```python
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib', '/usr/l

```  

*Changed in version 3.6:* Accepts a *path-like object*.

```python
os.path.dirname(path)
```

Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function `split()`.

*Changed in version 3.6:* Accepts a *path-like object.*
os.path.exists(path)

Return True if path refers to an existing path or an open file descriptor. Returns False for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute os.stat() on the requested file, even if the path physically exists.

*Changed in version 3.3:* path can now be an integer: True is returned if it is an open file descriptor, False otherwise.

*Changed in version 3.6:* Accepts a path-like object.

os.path.lexists(path)

Return True if path refers to an existing path. Returns True for broken symbolic links. Equivalent to exists() on platforms lacking os.lstat().

*Changed in version 3.6:* Accepts a path-like object.

os.path.expanduser(path)

On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that user’s home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user’s home directory is looked up in the password directory through the built-in module pwd. An initial ~user is looked up directly in the password directory.

On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by stripping the last directory component from the created user path derived above.
If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.expandvars(path)
```

Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable `name`. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.getatime(path)
```

Return the time of last access of `path`. The return value is a floating point number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

```python
os.path.getmtime(path)
```

Return the time of last modification of `path`. The return value is a floating point number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.getctime(path)
```

Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like
Windows), is the creation time for `path`. The return value is a number giving the number of seconds since the epoch (see the `time` module). Raise `OSError` if the file does not exist or is inaccessible.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.getsize(path)
```
Return the size, in bytes, of `path`. Raise `OSError` if the file does not exist or is inaccessible.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.isabs(path)
```
Return `True` if `path` is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.isfile(path)
```
Return `True` if `path` is an existing regular file. This follows symbolic links, so both `islink()` and `isfile()` can be true for the same path.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.isdir(path)
```
Return `True` if `path` is an existing directory. This follows symbolic links, so both `islink()` and `isdir()` can be true for the same path.

*Changed in version 3.6:* Accepts a path-like object.
os.path.islink(path)
Return True if path refers to an existing directory entry that is a symbolic link. Always False if symbolic links are not supported by the Python runtime.

*Changed in version 3.6:* Accepts a path-like object.

os.path.ismount(path)
Return True if pathname path is a mount point: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether path's parent, path/.., is on a different device than path, or whether path/.. and path point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. It is not able to reliably detect bind mounts on the same filesystem. On Windows, a drive letter root and a share UNC are always mount points, and for any other path GetVolumePathName is called to see if it is different from the input path.

*New in version 3.4:* Support for detecting non-root mount points on Windows.

*Changed in version 3.6:* Accepts a path-like object.

os.path.join(path, *paths)
Join one or more path components intelligently. The return value is the concatenation of path and any members of *paths with exactly one directory separator (os.sep) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.
On Windows, the drive letter is not reset when an absolute path component (e.g., `r'\foo'`) is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive C: (c:foo), not c:\foo.

*Changed in version 3.6:* Accepts a path-like object for path and paths.

```python
os.path.normcase(path)
```

Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes. Raise a `TypeError` if the type of `path` is not `str` or `bytes` (directly or indirectly through the `os.PathLike` interface).

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.normpath(path)
```

Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/\B`, `A/./B` and `A/foo/../B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use `normcase()`.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.realpath(path)
```

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.relpath(path, start=os.curdir)
```

Return a relative filepath to `path` either from the current directory or from an optional `start` directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `start`.

`start` defaults to `os.curdir`.

**Availability:** Unix, Windows.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.samefile(path1, path2)
```

Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an `os.stat()` call on either pathname fails.

**Availability:** Unix, Windows.

*Changed in version 3.2:* Added Windows support.

*Changed in version 3.4:* Windows now uses the same implementation as all other platforms.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.sameopenfile(fp1, fp2)
```
Return True if the file descriptors `fp1` and `fp2` refer to the same file.

**Availability:** Unix, Windows.

*Changed in version 3.2:* Added Windows support.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.samestat(stat1, stat2)
```

Return True if the stat tuples `stat1` and `stat2` refer to the same file. These structures may have been returned by `os.fstat()`, `os.lstat()`, or `os.stat()`. This function implements the underlying comparison used by `samefile()` and `sameopenfile()`.

**Availability:** Unix, Windows.

*Changed in version 3.4:* Added Windows support.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.path.split(path)
```

Split the pathname `path` into a pair, `(head, tail)` where `tail` is the last pathname component and `head` is everything leading up to that. The `tail` part will never contain a slash; if `path` ends in a slash, `tail` will be empty. If there is no slash in `path`, `head` will be empty. If `path` is empty, both `head` and `tail` are empty. Trailing slashes are stripped from `head` unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as `path` (but the strings may differ). Also see the functions `dirname()` and `basename()`.
Changed in version 3.6: Accepts a path-like object.

**os.path.splitdrive**(path)

Split the pathname path into a pair (drive, tail) where drive is either a mount point or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon. e.g. splitdrive("c:/dir") returns ("c:", "/dir")

If the path contains a UNC path, drive will contain the host name and share, up to but not including the fourth separator. e.g. splitdrive("//host/computer/dir") returns ("//host/computer", "/dir")

\[\text{Changed in version 3.6: Accepts a path-like object.}\]

**os.path.splitext**(path)

Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; splitext('.cshrc') returns ('.cshrc', '').

\[\text{Changed in version 3.6: Accepts a path-like object.}\]

**os.path.supports_unicode_filenames**

True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).
fileinput — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see open().

The typical use is:

```python
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If a filename is '-', it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first argument to input(). A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the mode parameter in the call to input() or FileInput. If an I/O error occurs during opening or reading a file, OSError is raised.

Changed in version 3.3: IOError used to be raised; it is now an alias of OSError.

If sys.stdin is used more than once, the second and further use will
return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using \texttt{sys.stdin.seek(0)}).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the \texttt{openhook} parameter to \texttt{fileinput.input()} or \texttt{FileInput()}. The hook must be a function that takes two arguments, \texttt{filename} and \texttt{mode}, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

\begin{verbatim}
fileinput.input(files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)
\end{verbatim}

Create an instance of the \texttt{FileInput} class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the \texttt{FileInput} class.

The \texttt{FileInput} instance can be used as a context manager in the \texttt{with} statement. In this example, \texttt{input} is closed after the \texttt{with} statement is exited, even if an exception occurs:

\begin{verbatim}
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
\end{verbatim}
Changed in version 3.2: Can be used as a context manager.

Deprecated since version 3.6, will be removed in version 3.8: The bufsize parameter.

The following functions use the global state created by fileinput.input(); if there is no active state, RuntimeError is raised.

fileinput.**filename**()

Return the name of the file currently being read. Before the first line has been read, returns None.

fileinput.**fileno**()

Return the integer “file descriptor” for the current file. When no file is opened (before the first line and between files), returns -1.

fileinput.**lineno**()

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line.

fileinput.**filelineno**()

Return the line number in the current file. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.**isfirstline**()

Returns true if the line just read is the first line of its file, otherwise returns false.

fileinput.**isstdin**()
Returns true if the last line was read from `sys.stdin`, otherwise returns false.

`fileinput.nextfile()`

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

`fileinput.close()`

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

```python
class fileinput.FileInput(files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)
```

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With `mode` you can specify which file mode will be passed to `open()`. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`. 
The openhook, when given, must be a function that takes two arguments, filename and mode, and returns an accordingly opened file-like object. You cannot use inplace and openhook together.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, input is closed after the with statement is exited, even if an exception occurs:

```python
with FileInput(files=('spam.txt', 'eggs.txt')) as process(input)
```

*Changed in version 3.2:* Can be used as a context manager.

*Deprecated since version 3.4:* The `'rU'` and `'U'` modes.

*Deprecated since version 3.6,* will be removed in version 3.8:* The bufsize parameter.

**Optional in-place filtering:** if the keyword argument `inplace=True` is passed to `fileinput.input()` or to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the `backup` parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

`fileinput.hook_compressed(filename, mode)`

Transparently opens files compressed with gzip and bzip2
(recognized by the extensions '.gz' and '.bz2') using the gzip and bz2 modules. If the filename extension is not '.gz' or '.bz2', the file is opened normally (ie, using open() without any decompression).

Usage example: fi = fileinput.FileInput(openhook=fileinput.hook_compressed)

fileinput.hook_encoded(encoding, errors=None)

Returns a hook which opens each file with open(), using the given encoding and errors to read the file.

Usage example: fi = fileinput.FileInput(openhook=fileinput.hook_encoded(8, "surrogateescape"))

Changed in version 3.6: Added the optional errors parameter.
stat — Interpreting stat() results

Source code: Lib/stat.py

The `stat` module defines constants and functions for interpreting the results of `os.stat()`, `os.fstat()` and `os.lstat()` (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

*Changed in version 3.4:* The stat module is backed by a C implementation.

The `stat` module defines the following functions to test for specific file types:

- `stat.S_ISDIR(mode)`
  Return non-zero if the mode is from a directory.

- `stat.S_ISCHR(mode)`
  Return non-zero if the mode is from a character special device file.

- `stat.S_ISBLK(mode)`
  Return non-zero if the mode is from a block special device file.

- `stat.S_ISREG(mode)`
  Return non-zero if the mode is from a regular file.

- `stat.S_ISFIFO(mode)`
  Return non-zero if the mode is from a FIFO (named pipe).
stat.\texttt{S\_ISLNK}(mode)
Return non-zero if the mode is from a symbolic link.

stat.\texttt{S\_ISSOCK}(mode)
Return non-zero if the mode is from a socket.

stat.\texttt{S\_ISDOOR}(mode)
Return non-zero if the mode is from a door.

\textit{New in version 3.4.}

stat.\texttt{S\_ISPORT}(mode)
Return non-zero if the mode is from an event port.

\textit{New in version 3.4.}

stat.\texttt{S\_ISWHT}(mode)
Return non-zero if the mode is from a whiteout.

\textit{New in version 3.4.}

Two additional functions are defined for more general manipulation of the file's mode:

stat.\texttt{S\_IMODE}(mode)
Return the portion of the file's mode that can be set by \texttt{os.chmod()}—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.\texttt{S\_IFMT}(mode)
Return the portion of the file's mode that describes the file type (used by the \texttt{S\_IS*()} functions above).

Normally, you would use the \texttt{os.path.is*()} functions for testing the
type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn't handled by `os.path`, like the tests for block and character devices.

Example:

```python
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted
    at top, calling the callback function for each regular
    file found.'''
    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a
human readable string:

**stat.**`filemode(mode)`

Convert a file’s mode to a string of the form ‘-rwxrwxrwx’.

*New in version 3.3.*

*Changed in version 3.4:* The function supports `S_IFDOOR`, `S_IFPORT` and `S_IFWHT`.

All the variables below are simply symbolic indexes into the 10-tuple returned by `os.stat()`, `os.fstat()` or `os.lstat()`.

**stat.ST_MODE**

Inode protection mode.

**stat.ST_INO**

Inode number.

**stat.ST_DEV**

Device inode resides on.

**stat.ST_NLINK**

Number of links to the inode.

**stat.ST_UID**

User id of the owner.

**stat.ST_GID**

Group id of the owner.

**stat.ST_SIZE**

Size in bytes of a plain file; amount of data waiting on some special files.
stat.**ST_ATIME**
  Time of last access.

stat.**ST_MTIME**
  Time of last modification.

stat.**ST_CTIME**
  The “ctime” as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of “file size” changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the “size” is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()`; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the **ST_MODE** field.

Use of the functions above is more portable than use of the first set of flags:

stat.**S_IFSOCK**
  Socket.

stat.**S_IFLNK**
  Symbolic link.
stat. **S_IFREG**
   Regular file.

stat. **S_IFBLK**
   Block device.

stat. **S_IFDIR**
   Directory.

stat. **S_IFCHR**
   Character device.

stat. **S_IFIFO**
   FIFO.

stat. **S_IFDOOR**
   Door.

   *New in version 3.4.*

stat. **S_IFPORT**
   Event port.

   *New in version 3.4.*

stat. **S_IFWHT**
   Whiteout.

   *New in version 3.4.*

**Note:** **S_IFDOOR**, **S_IFPORT** or **S_IFWHT** are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of
os.chmod():

stat.S_ISUID
Set UID bit.

stat.S_ISGID
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the S_ISGID bit set. For a file that does not have the group execution bit (S_IXGRP) set, the set-group-ID bit indicates mandatory file/record locking (see also S_ENFMT).

stat.S_ISVTX
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.S_IRWXU
Mask for file owner permissions.

stat.S_IRUSR
Owner has read permission.

stat.S_IWUSR
Owner has write permission.

stat.S_IXUSR
Owner has execute permission.

stat.S_IRWXG
Mask for group permissions.
stat. **S_IRGRP**
   Group has read permission.

stat. **S_IWGRP**
   Group has write permission.

stat. **S_IXGRP**
   Group has execute permission.

stat. **S_IRWXO**
   Mask for permissions for others (not in group).

stat. **S_IROTH**
   Others have read permission.

stat. **S_IWOTH**
   Others have write permission.

stat. **S_IXOTH**
   Others have execute permission.

stat. **S_ENFMT**
   System V file locking enforcement. This flag is shared with 
   **S_ISGID**: file/record locking is enforced on files that do not have 
   the group execution bit (**S_IXGRP**) set.

stat. **S_IREAD**
   Unix V7 synonym for **S_IRUSR**.

stat. **S_IWRITE**
   Unix V7 synonym for **S_IWUSR**.

stat. **S_IEXEC**
Unix V7 synonym for S_IXUSR.

The following flags can be used in the flags argument of os.chflags():

stat.**UF_NODUMP**
Do not dump the file.

stat.**UF_IMMUTABLE**
The file may not be changed.

stat.**UF_APPEND**
The file may only be appended to.

stat.**UF_OPAQUE**
The directory is opaque when viewed through a union stack.

stat.**UF_NOUNLINK**
The file may not be renamed or deleted.

stat.**UF_COMPRESSED**
The file is stored compressed (Mac OS X 10.6+).

stat.**UF_HIDDEN**
The file should not be displayed in a GUI (Mac OS X 10.5+).

stat.**SF_ARCHIVED**
The file may be archived.

stat.**SF_IMMUTABLE**
The file may not be changed.

stat.**SF_APPEND**
The file may only be appended to.
stat. **SF_NOUNLINK**

The file may not be renamed or deleted.

stat. **SF_SNAPSHOT**

The file is a snapshot file.

See the *BSD or Mac OS systems man page* `chflags(2)` for more information.

On Windows, the following file attribute constants are available for use when testing bits in the `st_file_attributes` member returned by `os.stat()`. See the Windows API documentation for more detail on the meaning of these constants.

stat. **FILE_ATTRIBUTE_ARCHIVE**
stat. **FILE_ATTRIBUTE_COMPRESSED**
stat. **FILE_ATTRIBUTE_DEVICE**
stat. **FILE_ATTRIBUTE_DIRECTORY**
stat. **FILE_ATTRIBUTE_ENCRYPTED**
stat. **FILE_ATTRIBUTE_HIDDEN**
stat. **FILE_ATTRIBUTE_INTEGRITY_STREAM**
stat. **FILE_ATTRIBUTE_NORMAL**
stat. **FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**
stat. **FILE_ATTRIBUTE_NO_SCRUB_DATA**
stat. **FILE_ATTRIBUTE_OFFLINE**
stat. **FILE_ATTRIBUTE_READONLY**
stat. **FILE_ATTRIBUTE_REPARSE_POINT**
stat. **FILE_ATTRIBUTE_SPARSE_FILE**
stat. **FILE_ATTRIBUTE_SYSTEM**
stat. **FILE_ATTRIBUTE_TEMPORARY**
stat. **FILE_ATTRIBUTE_VIRTUAL**

*New in version 3.5.*
**filecmp — File and Directory Comparisons**

**Source code:** Lib/filecmp.py

The `filecmp` module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the `difflib` module.

The `filecmp` module defines the following functions:

`filecmp.cmp(f1, f2, shallow=True)`

Compare the files named `f1` and `f2`, returning `True` if they seem equal, `False` otherwise.

If `shallow` is true, files with identical `os.stat()` signatures are taken to be equal. Otherwise, the contents of the files are compared.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the `os.stat()` information for the file changes. The entire cache may be cleared using `clear_cache()`.

`filecmp.cmpfiles(dir1, dir2, common, shallow=True)`

Compare the files in the two directories `dir1` and `dir2` whose names are given by `common`. 
Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don't, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don’t exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. 'c' and 'd/e' will each be in one of the three returned lists.

**filecmp.clear_cache()**

Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

*New in version 3.4.*
The **dircmp class**

```python
class filecmp.dircmp(a, b, ignore=None, hide=None)
```

Construct a new directory comparison object, to compare the directories `a` and `b`. `ignore` is a list of names to ignore, and defaults to `filecmp.DEFAULT_IGNORES`. `hide` is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The **dircmp class** compares files by doing *shallow* comparisons as described for `filecmp.cmp()`.

The **dircmp** class provides the following methods:

- **report()**
  - Print (to `sys.stdout`) a comparison between `a` and `b`.

- **report_partial_closure()**
  - Print a comparison between `a` and `b` and common immediate subdirectories.

- **report_full_closure()**
  - Print a comparison between `a` and `b` and common subdirectories (recursively).

The **dircmp** class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.
left
   The directory a.

right
   The directory b.

left_list
   Files and subdirectories in a, filtered by hide and ignore.

right_list
   Files and subdirectories in b, filtered by hide and ignore.

common
   Files and subdirectories in both a and b.

left_only
   Files and subdirectories only in a.

right_only
   Files and subdirectories only in b.

common_dirs
   Subdirectories in both a and b.

common_files
   Files in both a and b.

common_funny
   Names in both a and b, such that the type differs between the directories, or names for which os.stat() reports an error.

same_files
   Files which are identical in both a and b, using the class’s file comparison operator.
**diff_files**
Files which are in both a and b, whose contents differ according to the class's file comparison operator.

**funny_files**
Files which are in both a and b, but could not be compared.

**subdirs**
A dictionary mapping names in common_dirs to dircmp objects.

`filecmp.DEFAULT_IGNORES`

*New in version 3.4.*

List of directories ignored by dircmp by default.

Here is a simplified example of using the subdirs attribute to search recursively through two directories to show common different files:

```python
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" %
...             (name, dcmp.left, dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```
This module creates temporary files and directories. It works on all supported platforms. `TemporaryFile`, `NamedTemporaryFile`, `TemporaryDirectory`, and `SpooledTemporaryFile` are high-level interfaces which provide automatic cleanup and can be used as context managers. `mkstemp()` and `mkdtemp()` are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

```
tempfile.**TemporaryFile**(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)
```

Return a file-like object that can be used as a temporary storage area. The file is created securely, using the same rules as `mkstemp()`. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do
not support this; your code should not rely on a temporary file
created using this function having or not having a visible name in
the file system.

The resulting object can be used as a context manager (see
Examples). On completion of the context or destruction of the file
object the temporary file will be removed from the filesystem.

The mode parameter defaults to \texttt{\textquotesingle}w+b\textquotesingle so that the file created can
be read and written without being closed. Binary mode is used so
that it behaves consistently on all platforms without regard for the
data that is stored. \textit{buffering}, \textit{encoding} and \textit{newline} are interpreted
as for \texttt{open}().

The \texttt{dir}, \texttt{prefix} and \texttt{suffix} parameters have the same meaning and
defaults as with \texttt{mkstemp}().

The returned object is a true file object on POSIX platforms. On
other platforms, it is a file-like object whose \texttt{file} attribute is the
underlying true file object.

The \texttt{os.O_TMPFILE} flag is used if it is available and works (Linux-
specific, requires Linux kernel 3.11 or later).

\textit{Changed in version 3.5}: The \texttt{os.O_TMPFILE} flag is now used if
available.

\begin{verbatim}
tempfile.NamedTemporaryFile(mode='w+b',
buffering=None, encoding=None, newline=None, suffix=None,
prefix=None, dir=None, delete=True)
\end{verbatim}

This function operates exactly as \texttt{TemporaryFile()} does,
except that the file is guaranteed to have a visible name in the file
system (on Unix, the directory entry is not unlinked). That name
can be retrieved from the name attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If delete is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

```python
tempfile.SpooledTemporaryFile(max_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)
```

This function operates exactly as TemporaryFile() does, except that data is spooled in memory until the file size exceeds max_size, or until the file’s fileno() method is called, at which point the contents are written to disk and operation proceeds as with TemporaryFile().

The resulting file has one additional method, rollover(), which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose _file attribute is either an io.BytesIO or io.StringIO object (depending on whether binary or text mode was specified) or a true file object, depending on whether rollover() has been called. This file-like object can be used in a with statement, just like a normal file.

*Changed in version 3.3:* the truncate method now accepts a size argument.

```python
tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None)
```
This function securely creates a temporary directory using the same rules as `mkdtemp()`. The resulting object can be used as a context manager (see Examples). On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the name attribute of the returned object. When the returned object is used as a context manager, the name will be assigned to the target of the as clause in the `with` statement, if there is one.

The directory can be explicitly cleaned up by calling the `cleanup()` method.

*New in version 3.2.*

tempfile. **mkstemp**(suffix=None, prefix=None, dir=None, text=False)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the `os.O_EXCL` flag for `os.open()`. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike `TemporaryFile()`, the user of `mkstemp()` is responsible for deleting the temporary file when done with it.

If `suffix` is not `None`, the file name will end with that suffix, otherwise there will be no suffix. `mkstemp()` does not put a dot between the file name and the suffix; if you need one, put it at the
beginning of suffix.

If prefix is not None, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of getempprefix() or getempprefixb(), as appropriate.

If dir is not None, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the TMPDIR, TEMP or TMP environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via os.popen().

If any of suffix, prefix, and dir are not None, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass suffix=b''.

If text is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by os.open()) and the absolute pathname of that file, in that order.

Changed in version 3.5: suffix, prefix, and dir may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. suffix and prefix now accept and default to None to cause an appropriate default value to be used.
tempfile. **mkdtemp**(suffix=None, prefix=None, dir=None)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of **mkdtemp()** is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for **mkstemp()**.

**mkdtemp()** returns the absolute pathname of the new directory.

*Changed in version 3.5:* *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to **None** to cause an appropriate default value to be used.

tempfile. **gettempdir**()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   - On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.

5. As a last resort, the current working directory.
   The result of this search is cached, see the description of `tempdir` below.

`tempfile.gettempdirb()`
Same as `gettempdir()` but the return value is in bytes.

*New in version 3.5.*

`tempfile.gettempprefix()`
Return the filename prefix used to create temporary files. This does not contain the directory component.

`tempfile.gettempprefixb()`
Same as `gettempprefix()` but the return value is in bytes.

*New in version 3.5.*

The module uses a global variable to store the name of the directory used for temporary files returned by `gettempdir()`. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a `dir` argument which can be used to specify the directory and this is the recommended approach.

`tempfile.tempdir`
When set to a value other than `None`, this variable defines the default value for the `dir` argument to the functions defined in this module.

If `tempdir` is `None` (the default) at any call to any of the above functions except `gettempprefix()` it is initialized following the
algorithm described in `gettempdir()`.
Examples

Here are some examples of typical usage of the `tempfile` module:

```python
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
'b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
'b'Hello world!'

# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)

# directory and contents have been removed
```
Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the `mktemp()` function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to `mktemp()` and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by `mkstemp()` and the other functions described above.

```python
tempfile.mktemp(suffix='', prefix='tmp', dir=None)
```

*Deprecated since version 2.3:* Use `mkstemp()` instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The `prefix`, `suffix`, and `dir` arguments are similar to those of `mkstemp()`, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

**Warning:** Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. `mktemp()` usage can be replaced easily with `NamedTemporaryFile()`, passing it the `delete=False` parameter:

```python
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
```
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os.unlink(f.name)</code></td>
<td>Removes the file</td>
</tr>
<tr>
<td><code>os.path.exists(f.name)</code></td>
<td>Checks if the file exists</td>
</tr>
</tbody>
</table>

```
False
```
The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the `os.scandir()` and `fnmatch.fnmatch()` functions in concert, and not by actually invoking a subshell. Note that unlike `fnmatch.fnmatch()`, `glob` treats filenames beginning with a dot (.) as special cases. (For tilde and shell variable expansion, use `os.path.expanduser()` and `os.path.expandvars()`.)

For a literal match, wrap the meta-characters in brackets. For example, `'[?]']` matches the character `'?`.

See also: The `pathlib` module offers high-level path objects.

```
glob.glob(pathname, *, recursive=False)
```

Return a possibly-empty list of path names that match `pathname`, which must be a string containing a path specification. `pathname` can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

If `recursive` is true, the pattern `"**"` will match any files and zero or
more directories and subdirectories. If the pattern is followed by an `os.sep`, only directories and subdirectories match.

**Note:** Using the "**" pattern in large directory trees may consume an inordinate amount of time.

*Changed in version 3.5:* Support for recursive globs using "**".

`glob.iglob(pathname, *, recursive=False)`
Return an iterator which yields the same values as `glob()` without actually storing them all simultaneously.

`glob.escape(pathname)`
Escape all special characters ("?", "*", and "]"). This is useful if you want to match an arbitrary literal string that may have special characters in it. Special characters in drive/UNC sharepoints are not escaped, e.g. on Windows `escape('/\?/c:/Quo vadis?.txt')` returns '/\?/c:/Quo vadis[?].txt'.

*New in version 3.4.*

For example, consider a directory containing the following files: 1.gif, 2.txt, card.gif and a subdirectory sub which contains only the file 3.txt. `glob()` will produce the following results. Notice how any leading components of the path are preserved.

```python
>>> import glob
>>> glob.glob('.//[0-9].*')
[ './1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
```


```python
['2.txt', 'sub/3.txt']

>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

If the directory contains files starting with . they won't be matched by default. For example, consider a directory containing `card.gif` and `.card.gif`:

```python
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also:

**Module fnmatch**

Shell-style filename (not path) expansion
fnmatch — Unix filename pattern matching

Source code: Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are not the same as regular expressions (which are documented in the re module). The special characters used in shell-style wildcards are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches everything</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[seq]</td>
<td>matches any character in seq</td>
</tr>
<tr>
<td>[!seq]</td>
<td>matches any character not in seq</td>
</tr>
</tbody>
</table>

For a literal match, wrap the meta-characters in brackets. For example, '[-?]' matches the character '!?'.

Note that the filename separator ('/' on Unix) is not special to this module. See module glob for pathname expansion (glob uses filter() to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the * and ? patterns.

fnmatch.fnmatch(filename, pattern)

Test whether the filename string matches the pattern string, returning True or False. Both parameters are case-normalized
using `os.path.normcase()`. `fnmatchcase()` can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

This example will print all file names in the current directory with the extension `.txt`:

```python
import fnmatch
import os

for file in os.listdir('. '):
    if fnmatch.fnmatch(file, '*.txt '):
        print(file)
```

`fnmatch.fnmatchcase(filename, pattern)`

Test whether `filename` matches `pattern`, returning `True` or `False`; the comparison is case-sensitive and does not apply `os.path.normcase()`.

`fnmatch.filter(names, pattern)`

Return the subset of the list of `names` that match `pattern`. It is the same as `[n for n in names if fnmatch(n, pattern)]`, but implemented more efficiently.

`fnmatch.translate(pattern)`

Return the shell-style `pattern` converted to a regular expression for using with `re.match()`.

Example:

```python
>>> import fnmatch, re

>>> regex = fnmatch.translate('*.txt ')
>>> regex
```
'(?s:.*/\.txt)\Z'

```python
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

See also:

**Module glob**

Unix shell-style path expansion.
linecache — Random access to text lines

Source code: Lib/linecache.py

The linecache module allows one to get any line from a Python source file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the traceback module to retrieve source lines for inclusion in the formatted traceback.

The tokenize.open() function is used to open files. This function uses tokenize.detect_encoding() to get the encoding of the file; in the absence of an encoding token, the file encoding defaults to UTF-8.

The linecache module defines the following functions:

linecache.getline(filename, lineno, module_globals=None)

Get line lineno from file named filename. This function will never raise an exception — it will return '' on errors (the terminating newline character will be included for lines that are found).

If a file named filename is not found, the function will look for it in the module search path, sys.path, after first checking for a PEP 302 __loader__ in module_globals, in case the module was imported from a zipfile or other non-filesystem import source.

linecache.clearcache()

Clear the cache. Use this function if you no longer need lines from
files previously read using `getline()`.

`linecache.checkcache(filename=None)`

Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If `filename` is omitted, it will check all the entries in the cache.

`linecache.lazyload(filename, module_globals)`

Capture enough detail about a non-file-based module to permit getting its lines later via `getline()` even if `module_globals` is `None` in the later call. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely.

*New in version 3.5.*

Example:

```python
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys
'
```
shutil — High-level file operations

Source code: Lib/shutil.py

The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the os module.

Warning: Even the higher-level file copying functions (shutil.copy(), shutil.copy2()) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.
Directory and files operations

**shutil.copyfileobj**(*fsrc, fdst[, length]*)

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

**shutil.copyfile**(*src, dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst*. *src* and *dst* are path names given as strings. *dst* must be the complete target file name; look at *shutil.copy()* for a copy that accepts a target directory path. If *src* and *dst* specify the same file, *SameFileError* is raised.

The destination location must be writable; otherwise, an *OSError* exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

*Changed in version 3.3*: *IOError* used to be raised instead of *OSError*. Added *follow_symlinks* argument. Now returns *dst*.

*Changed in version 3.4*: Raise *SameFileError* instead of
Error. Since the former is a subclass of the latter, this change is backward compatible.

exception shutil.SameFileError

This exception is raised if source and destination in copyfile() are the same file.

New in version 3.4.

shutil.copymode(src, dst, *, follow_symlinks=True)

Copy the permission bits from src to dst. The file contents, owner, and group are unaffected. src and dst are path names given as strings. If follow_symlinks is false, and both src and dst are symbolic links, copymode() will attempt to modify the mode of dst itself (rather than the file it points to). This functionality is not available on every platform; please see copystat() for more information. If copymode() cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Changed in version 3.3: Added follow_symlinks argument.

shutil.copystat(src, dst, *, follow_symlinks=True)

Copy the permission bits, last access time, last modification time, and flags from src to dst. On Linux, copystat() also copies the “extended attributes” where possible. The file contents, owner, and group are unaffected. src and dst are path names given as strings.

If follow_symlinks is false, and src and dst both refer to symbolic links, copystat() will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the src symbolic link, and writing the information to the dst symbolic link.
Note: Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.

- If `os.chmod` in `os.supports_follow_symlinks` is True, `copystat()` can modify the permission bits of a symbolic link.
- If `os.utime` in `os.supports_follow_symlinks` is True, `copystat()` can modify the last access and modification times of a symbolic link.
- If `os.chflags` in `os.supports_follow_symlinks` is True, `copystat()` can modify the flags of a symbolic link. (os.chflags is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, `copystat()` will copy everything it can. `copystat()` never returns failure.

Please see `os.supports_followSymlinks` for more information.

*Changed in version 3.3:* Added `follow_symlinks` argument and support for Linux extended attributes.

`shutil.copy(src, dst, *, follow_symlinks=True)`

Copies the file `src` to the file or directory `dst`. `src` and `dst` should be strings. If `dst` specifies a directory, the file will be copied into `dst` using the base filename from `src`. Returns the path to the newly created file.

If `follow_symlinks` is false, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks` is true and `src` is a symbolic link, `dst` will be a copy of the file `src` refers to.

`copy()` copies the file data and the file's permission mode (see
os.chmod()). Other metadata, like the file’s creation and modification times, is not preserved. To preserve all file metadata from the original, use copy2() instead.

*Changed in version 3.3:* Added follow_symlinks argument. Now returns path to the newly created file.

**shutil.copy2(src, dst, *, follow_symlinks=True)**

Identical to copy() except that copy2() also attempts to preserve file metadata.

When follow_symlinks is false, and src is a symbolic link, copy2() attempts to copy all metadata from the src symbolic link to the newly-created dst symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, copy2() will preserve all the metadata it can; copy2() never returns failure.

copy2() uses copystat() to copy the file metadata. Please see copystat() for more information about platform support for modifying symbolic link metadata.

*Changed in version 3.3:* Added follow_symlinks argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

**shutil.ignore_patterns(*patterns)**

This factory function creates a function that can be used as a callable for copytree()’s ignore argument, ignoring files and directories that match one of the glob-style patterns provided. See the example below.

**shutil.copytree(src, dst, symlinks=False, ignore=None,**
Recursively copy an entire directory tree rooted at `src`, returning the destination directory. The destination directory, named by `dst`, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with `copystat()`, individual files are copied using `shutil.copy2()`.

If `symlinks` is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When `symlinks` is false, if the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional `ignore_dangling_symlinks` flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support `os.symlink()`.

If `ignore` is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the `ignore` callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.
If `copy_function` is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used.

*Changed in version 3.3:* Copy metadata when `symlinks` is false. Now returns `dst`.

*Changed in version 3.2:* Added the `copy_function` argument to be able to provide a custom copy function. Added the `ignore_dangling_symlinks` argument to silent dangling symlinks errors when `symlinks` is false.

```python
shutil.rmtree(path, ignore_errors=False, onerror=None)
```
Delete an entire directory tree; `path` must point to a directory (but not a symbolic link to a directory). If `ignore_errors` is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by `onerror` or, if that is omitted, they raise an exception.

**Note:** On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn’t be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If `onerror` is provided, it must be a callable that accepts three
parameters: function, path, and excinfo.

The first parameter, function, is the function which raised the exception; it depends on the platform and implementation. The second parameter, path, will be the path name passed to function. The third parameter, excinfo, will be the exception information returned by sys.exc_info(). Exceptions raised by onerror will not be caught.

Changed in version 3.3: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

rmtree.avoids_symlink_attacks

Indicates whether the current platform and implementation provides a symlink attack resistant version of rmtree(). Currently this is only true for platforms supporting fd-based directory access functions.

New in version 3.3.

shutil.move(src, dst, copy_function=copy2)

Recursively move a file or directory (src) to another location (dst) and return the destination.

If the destination is an existing directory, then src is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on the current filesystem, then os.rename() is used. Otherwise, src is copied to dst using copy_function and then removed. In case of symlinks, a new symlink pointing to the target of src will be created in or as dst and src will be removed.
If `copy_function` is given, it must be a callable that takes two arguments `src` and `dst`, and will be used to copy `src` to `dest` if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the `copy_function()`. The default `copy_function` is `copy2()`. Using `copy()` as the `copy_function` allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

*Changed in version 3.3:* Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU’s `mv`. Now returns `dst`.

*Changed in version 3.5:* Added the `copy_function` keyword argument.

**shutil.disk_usage(path)**

Return disk usage statistics about the given path as a named tuple with the attributes `total`, `used` and `free`, which are the amount of total, used and free space, in bytes. On Windows, `path` must be a directory; on Unix, it can be a file or directory.

*New in version 3.3.*

**Availability:** Unix, Windows.

**shutil.chown(path, user=None, group=None)**

Change owner `user` and/or `group` of the given `path`.

`user` can be a system user name or a uid; the same applies to `group`. At least one argument is required.

See also `os.chown()`, the underlying function.
Availability: Unix.

New in version 3.3.

```
shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)
```

Return the path to an executable which would be run if the given `cmd` was called. If no `cmd` would be called, return None.

`mode` is a permission mask passed to `os.access()`, by default determining if the file exists and executable.

When no `path` is specified, the results of `os.environ()` are used, returning either the “PATH” value or a fallback of `os.defpath`.

On Windows, the current directory is always prepended to the `path` whether or not you use the default or provide your own, which is the behavior the command shell uses when finding executables. Additionally, when finding the `cmd` in the `path`, the `PATHEXT` environment variable is checked. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the `path` directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

New in version 3.3.

```
exception shutil.Error
```

This exception collects exceptions that are raised during a multi-file operation. For `copytree()`, the exception argument is a list of 3-tuples (`srcname`, `dstname`, `exception`).
copytree example

This example is the implementation of the `copytree()` function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```python
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
        # XXX What about devices, sockets etc.? except OSError as why:
            errors.append((srcname, dstname, str(why))
        # catch the Error from the recursive copytree so that we can # continue with other files except Error as err:
            errors.extend(err.args[0])
        try:
            copystat(src, dst)
        except OSError as why:
            # can't copy file access times on Windows
            if why.winerror is None:
                errors.extend([(src, dst, str(why))])
        if errors:
```

```
Another example that uses the `ignore_patterns()` helper:

```python
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns)
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the `ignore` argument to add a logging call:

```python
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []  # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onerror callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```python
import os, stat
import shutil

def remove_readonly(func, path, _):
```
"Clear the readonly bit and reattempt the removal"

```
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```
Archiving operations

New in version 3.2.

Changed in version 3.5: Added support for the xztar format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the zipfile and tarfile modules.

```
shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]])
```

Create an archive file (such as zip or tar) and return its name.

- `base_name` is the name of the file to create, including the path, minus any format-specific extension. `format` is the archive format: one of “zip” (if the zlib module is available), “tar”, “gztar” (if the zlib module is available), “bztar” (if the bz2 module is available), or “xztar” (if the lzma module is available).

- `root_dir` is a directory that will be the root directory of the archive; for example, we typically chdir into `root_dir` before creating the archive.

- `base_dir` is the directory where we start archiving from; i.e. `base_dir` will be the common prefix of all files and directories in the archive.

- `root_dir` and `base_dir` both default to the current directory.

If `dry_run` is true, no archive is created, but the operations that would be executed are logged to `logger`.

`owner` and `group` are used when creating a tar archive. By default,
uses the current owner and group.

*logger* must be an object compatible with [PEP 282](https://www.python.org/dev/peps/pep-0282/), usually an instance of `logging.Logger`.

The *verbose* argument is unused and deprecated.

**shutil.get_archive_formats()**

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default, `shutil` provides these formats:

- `zip`: ZIP file (if the `zlib` module is available).
- `tar`: uncompressed tar file.
- `gztar`: gzip'ed tar-file (if the `zlib` module is available).
- `bztar`: bzip2'ed tar-file (if the `bz2` module is available).
- `xztar`: xz'ed tar-file (if the `lzma` module is available).

You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

**shutil.register_archive_format(name, function[, extra_args[, description]])**

Register an archiver for the format *name*.

*function* is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to `os.curdir`) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in `make_archive()`).

If given, *extra_args* is a sequence of `(name, value)` pairs that
will be used as extra keywords arguments when the archiver callable is used.

*description* is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty string.

`shutil.unregister_archive_format(name)`
Remove the archive format `name` from the list of supported formats.

`shutil.unpack_archive(filename[, extract_dir[, format]])`
Unpack an archive. `filename` is the full path of the archive.

`extract_dir` is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

`format` is the archive format: one of “zip”, “tar”, “gztar”, “bztar”, or “xztar”. Or any other format registered with `register_unpack_format()`. If not provided, `unpack_archive()` will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a `ValueError` is raised.

*Changed in version 3.7:* Accepts a path-like object for `filename` and `extract_dir`.

`shutil.register_unpack_format(name, extensions, function[, extra_args[, description]])`
 Registers an unpack format. `name` is the name of the format and `extensions` is a list of extensions corresponding to the format, like `.zip` for Zip files.

`function` is the callable that will be used to unpack archives. The
callable will receive the path of the archive, followed by the directory the archive must be extracted to.

When provided, `extra_args` is a sequence of `(name, value)` tuples that will be passed as keywords arguments to the callable.

`description` can be provided to describe the format, and will be returned by the `get_unpack_formats()` function.

```python
shutil.unregister_unpack_format(name)
```

Unregister an unpack format. `name` is the name of the format.

```python
shutil.get_unpack_formats()
```

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

By default `shutil` provides these formats:

- `zip`: ZIP file (unpacking compressed files works only if the corresponding module is available).
- `tar`: uncompressed tar file.
- `gztar`: gzip’ed tar-file (if the `zlib` module is available).
- `bztar`: bzip2’ed tar-file (if the `bz2` module is available).
- `xztar`: xz’ed tar-file (if the `lzma` module is available).

You can register new formats or provide your own unpacker for any existing formats, by using `register_unpack_format()`.

**Archiving example**

In this example, we create a gzip’ed tar-file archive containing all files found in the `.ssh` directory of the user:
```python
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(archive_name, 'gztar', root_dir, '/Users/tarek/myarchive.tar.gz')
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
  drwx------  tarek/staff      0 2010-02-01 16:23:40 ./
  -rw-r--r--  tarek/staff    609 2008-06-09 13:26:54 ./authorized_keys
  -rwxr-xr-x  tarek/staff     65 2008-06-09 13:26:54 ./config
  -rwx------  tarek/staff    668 2008-06-09 13:26:54 ./id_dsa
  -rwxr-xr-x  tarek/staff    609 2008-06-09 13:26:54 ./id_dsa.pub
  -rw-------  tarek/staff   1675 2008-06-09 13:26:54 ./id_rsa
  -rw-r--r--  tarek/staff    397 2008-06-09 13:26:54 ./id_rsa.pub
  -rw-r--r--  tarek/staff  37192 2010-02-06 18:23:10 ./known_hosts
```
Querying the size of the output terminal

```
shutil.get_terminal_size(fallback=(columns, lines))
```

Get the size of the terminal window.

For each of the two dimensions, the environment variable, COLUMNS and LINES respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When COLUMNS or LINES is not defined, which is the common case, the terminal connected to `sys.__stdout__` is queried by invoking `os.get_terminal_size()`.

If the terminal size cannot be successfully queried, either because the system doesn’t support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)`, which is the default size used by many terminal emulators.

The value returned is a named tuple of type `os.terminal_size`.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

*New in version 3.3.*
macpath — Mac OS 9 path manipulation functions

Source code: Lib/macpath.py

Deprecated since version 3.7, will be removed in version 3.8.

This module is the Mac OS 9 (and earlier) implementation of the os.path module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: normcase(), normpath(), isabs(), join(), split(), isdir(), isfile(), walk(), exists(). For other functions available in os.path dummy counterparts are available.
Data Persistence

The modules described in this chapter support storing Python data in a persistent form on disk. The pickle and marshal modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings.

The list of modules described in this chapter is:

- **pickle** — Python object serialization
  - Relationship to other Python modules
    - Comparison with marshal
    - Comparison with json
  - Data stream format
  - Module Interface
  - What can be pickled and unpickled?
  - Pickling Class Instances
    - Persistence of External Objects
    - Dispatch Tables
    - Handling Stateful Objects
  - Restricting Globals
  - Performance
  - Examples
- **copyreg** — Register pickle support functions
  - Example
- **shelve** — Python object persistence
  - Restrictions
  - Example
- **marshal** — Internal Python object serialization
- **dbm** — Interfaces to Unix “databases”
- `dbm.gnu` — GNU’s reinterpretation of `dbm`
- `dbm.ndbm` — Interface based on `ndbm`
- `dbm.dumb` — Portable DBM implementation

- `sqlite3` — DB-API 2.0 interface for SQLite databases
  - Module functions and constants
  - Connection Objects
  - Cursor Objects
  - Row Objects
  - Exceptions
  - SQLite and Python types
    - Introduction
    - Using adapters to store additional Python types in SQLite databases
      - Letting your object adapt itself
      - Registering an adapter callable
      - Converting SQLite values to custom Python types
    - Default adapters and converters
  - Controlling Transactions
  - Using `sqlite3` efficiently
    - Using shortcut methods
    - Accessing columns by name instead of by index
    - Using the connection as a context manager
  - Common issues
    - Multithreading
The `pickle` module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling," [1] or "flattening"; however, to avoid confusion, the terms used here are "pickling" and "unpickling".

**Warning:** The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.
Relationship to other Python modules

Comparison with marshal

Python has a more primitive serialization module called `marshal`, but in general `pickle` should always be the preferred way to serialize Python objects. `marshal` exists primarily to support Python’s `.pyc` files.

The `pickle` module differs from `marshal` in several significant ways:

- The `pickle` module keeps track of the objects it has already serialized, so that later references to the same object won’t be serialized again. `marshal` doesn’t do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by `marshal`, and in fact, attempting to `marshal` recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. `pickle` stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- `marshal` cannot be used to serialize user-defined classes and their instances. `pickle` can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The `marshal` serialization format is not guaranteed to be
portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The pickle serialization format is guaranteed to be backwards compatible across Python releases provided a compatible pickle protocol is chosen and pickling and unpickling code deals with Python 2 to Python 3 type differences if your data is crossing that unique breaking change language boundary.

Comparison with json

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation):

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to utf-8), while pickle is a binary serialization format;
- JSON is human-readable, while pickle is not;
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python’s introspection facilities; complex cases can be tackled by implementing specific object APIs).

See also: The json module: a standard library module allowing JSON serialization and deserialization.
Data stream format

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON or XDR (which can’t represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the `pickle` data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently compress pickled data.

The module `pickletools` contains tools for analyzing data streams generated by `pickle`. `pickletools` source code has extensive comments about opcodes used by pickle protocols.

There are currently 5 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- Protocol version 0 is the original “human-readable” protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is an old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes. Refer to PEP 307 for information about improvements brought by protocol 2.
- Protocol version 3 was added in Python 3.0. It has explicit support for `bytes` objects and cannot be unpickled by Python 2.x. This is the default protocol, and the recommended protocol when compatibility with other Python 3 versions is required.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data
format optimizations. Refer to PEP 3154 for information about improvements brought by protocol 4.

**Note:** Serialization is a more primitive notion than persistence; although pickle reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The shelf module provides a simple interface to pickle and unpickle objects on DBM-style database files.
Module Interface

To serialize an object hierarchy, you simply call the `dumps()` function. Similarly, to de-serialize a data stream, you call the `loads()` function. However, if you want more control over serialization and de-serialization, you can create a Pickler or an Unpickler object, respectively.

The `pickle` module provides the following constants:

**pickle.HIGHEST_PROTOCOL**

An integer, the highest `protocol version` available. This value can be passed as a `protocol` value to functions `dump()` and `dumps()` as well as the Pickler constructor.

**pickle.DEFAULT_PROTOCOL**

An integer, the default `protocol version` used for pickling. May be less than `HIGHEST_PROTOCOL`. Currently the default protocol is 3, a new protocol designed for Python 3.

The `pickle` module provides the following functions to make the pickling process more convenient:

**pickle.dump(obj, file, protocol=None, *, fix_imports=True)**

Write a pickled representation of `obj` to the open file object `file`. This is equivalent to `Pickler(file, protocol).dump(obj)`.

The optional `protocol` argument, an integer, tells the pickler to use the given protocol; supported protocols are 0 to `HIGHEST_PROTOCOL`. If not specified, the default is `DEFAULT_PROTOCOL`. If a negative number is specified,
HIGHEST_PROTOCOL is selected.

The file argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an io.BytesIO instance, or any other custom object that meets this interface.

If fix_imports is true and protocol is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

```
pickle.dumps(obj, protocol=None, *, fix_imports=True)
```

Return the pickled representation of the object as a bytes object, instead of writing it to a file.

Arguments protocol and fix_imports have the same meaning as in dump().

```
pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")
```

Read a pickled object representation from the open file object file and return the reconstituted object hierarchy specified therein. This is equivalent to Unpickler(file).load().

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

The argument file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus file can be an on-disk file opened for binary reading, an io.BytesIO object, or any other custom object that meets this interface.
Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to ‘ASCII’ and ‘strict’, respectively. The *encoding* can be ‘bytes’ to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of `datetime`, `date` and `time` pickled by Python 2.

```
import pickle

pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")
```

Read a pickled object hierarchy from a `bytes` object and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to ‘ASCII’ and ‘strict’, respectively. The *encoding* can be ‘bytes’ to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of `datetime`, `date` and `time` pickled by Python 2.

The *pickle* module defines three exceptions:
exception pickle.**PickleError**

Common base class for the other pickling exceptions. It inherits `Exception`.

exception pickle.**PicklingError**

Error raised when an unpicklable object is encountered by `Pickler`. It inherits `PickleError`.

Refer to [What can be pickled and unpickled?](#) to learn what kinds of objects can be pickled.

exception pickle.**UnpicklingError**

Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits `PickleError`.

Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) `AttributeError`, `EOFError`, `ImportError`, and `IndexError`.

The pickle module exports two classes, `Pickler` and `Unpickler`:

```python
class pickle.**Pickler**(file, protocol=None, *, fix_imports=True)

This takes a binary file for writing a pickle data stream.

The optional `protocol` argument, an integer, tells the pickler to use the given protocol; supported protocols are 0 to `HIGHEST_PROTOCOL`. If not specified, the default is `DEFAULT_PROTOCOL`. If a negative number is specified, `HIGHEST_PROTOCOL` is selected.

The `file` argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an `io.BytesIO` instance, or any other custom object that
meets this interface.

If fix_imports is true and protocol is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

**dump**(*obj*)

Write a pickled representation of *obj* to the open file object given in the constructor.

**persistent_id**(*obj*)

Do nothing by default. This exists so a subclass can override it.

If persistent_id() returns None, *obj* is pickled as usual. Any other value causes Pickler to emit the returned value as a persistent ID for *obj*. The meaning of this persistent ID should be defined by Unpickler.persistent_load(). Note that the value returned by persistent_id() cannot itself have a persistent ID.

See Persistence of External Objects for details and examples of uses.

**dispatch_table**

A pickler object's dispatch table is a registry of reduction functions of the kind which can be declared using copyreg.pickle(). It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a __reduce__() method.

By default, a pickler object will not have a dispatch_table attribute, and it will instead use the global dispatch table.
managed by the `copyreg` module. However, to customize the pickling for a specific pickler object one can set the `dispatch_table` attribute to a dict-like object. Alternatively, if a subclass of `Pickler` has a `dispatch_table` attribute then this will be used as the default dispatch table for instances of that class.

See `Dispatch Tables` for usage examples.

`New in version 3.3.`

**fast**

Deprecated. Enable fast mode if set to a true value. The fast mode disables the usage of memo, therefore speeding the pickling process by not generating superfluous PUT opcodes. It should not be used with self-referential objects, doing otherwise will cause `Pickler` to recurse infinitely.

Use `pickletools.optimize()` if you need more compact pickles.

```python
class pickle.Unpickler(file, *, fix_imports=True, encoding="ASCII", errors="strict")
```

This takes a binary file for reading a pickle data stream.

The protocol version of the pickle is detected automatically, so no protocol argument is needed.

The argument `file` must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus `file` can be an on-disk file object opened for binary reading, an `io.BytesIO` object, or any other custom object that meets this interface.
Optional keyword arguments are `fix_imports`, `encoding` and `errors`, which are used to control compatibility support for pickle stream generated by Python 2. If `fix_imports` is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The `encoding` and `errors` tell pickle how to decode 8-bit string instances pickled by Python 2; these default to ‘ASCII’ and ‘strict’, respectively. The `encoding` can be ‘bytes’ to read these 8-bit string instances as bytes objects.

**load()**
Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled object’s representation are ignored.

**persistent_load(pid)**
Raise an `UnpicklingError` by default.

If defined, `persistent_load()` should return the object specified by the persistent ID `pid`. If an invalid persistent ID is encountered, an `UnpicklingError` should be raised.

See Persistence of External Objects for details and examples of uses.

**find_class(module, name)**
Import `module` if necessary and return the object called `name` from it, where the `module` and `name` arguments are `str` objects. Note, unlike its name suggests, `find_class()` is also used for finding functions.

Subclasses may override this to gain control over what type of objects and how they can be loaded, potentially reducing
security risks. Refer to Restricting Globals for details.
What can be pickled and unpickled?

The following types can be pickled:

- None, True, and False
- integers, floating point numbers, complex numbers
- strings, bytes, bytearrays
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module (using `def`, not `lambda`)
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose `__dict__` or the result of calling `__getstate__()` is picklable (see section Pickling Class Instances for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RecursionError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by “fully qualified” name reference, not by value. [2] This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function’s code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised. [3]
Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class’s code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment:

```python
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class’s code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.
Pickling Class Instances

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its `__init__()` method is usually not invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour:

```python
def save(obj):
    return (obj.__class__, obj.__dict__)
def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Classes can alter the default behaviour by providing one or several special methods:

`object.__getnewargs_ex__()`

In protocols 2 and newer, classes that implements the `__getnewargs_ex__()` method can dictate the values passed to the `__new__()` method upon unpickling. The method must return a pair `(args, kwargs)` where `args` is a tuple of positional arguments and `kwargs` a dictionary of named arguments for constructing the object. Those will be passed to the `__new__()`
method upon unpickling.

You should implement this method if the __new__() method of your class requires keyword-only arguments. Otherwise, it is recommended for compatibility to implement __getnewargs__().

*Changed in version 3.6:* __getnewargs_ex__() is now used in protocols 2 and 3.

object.__getnewargs__()

This method serves a similar purpose as __getnewargs_ex__(), but supports only positional arguments. It must return a tuple of arguments args which will be passed to the __new__() method upon unpickling.

__getnewargs__() will not be called if __getnewargs_ex__() is defined.

*Changed in version 3.6:* Before Python 3.6, __getnewargs__() was called instead of __getnewargs_ex__() in protocols 2 and 3.

object.__getstate__()

Classes can further influence how their instances are pickled; if the class defines the method __getstate__(), it is called and the returned object is pickled as the contents for the instance, instead of the contents of the instance’s dictionary. If the __getstate__() method is absent, the instance’s __dict__ is pickled as usual.

object.__setstate__(state)
Upon unpickling, if the class defines `__setstate__()`, it is called with the unpickled state. In that case, there is no requirement for the state object to be a dictionary. Otherwise, the pickled state must be a dictionary and its items are assigned to the new instance’s dictionary.

**Note:** If `__getstate__()` returns a false value, the `__setstate__()` method will not be called upon unpickling.

Refer to the section **Handling Stateful Objects** for more information about how to use the methods `__getstate__()` and `__setstate__()`.  

**Note:** At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__getnewargs__()` or `__getnewargs_ex__()` to establish such an invariant; otherwise, neither `__new__()` nor `__init__()` will be called.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the `__reduce__()` special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects. [4]

Although powerful, implementing `__reduce__()` directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., `__getnewargs_ex__()`, `__getstate__()` and `__setstate__()`) whenever possible. We will show, however, cases where using `__reduce__()` is the only
option or leads to more efficient pickling or both.

__reduce__()  
The interface is currently defined as follows. The __reduce__() method takes no argument and shall return either a string or preferably a tuple (the returned object is often referred to as the “reduce value”).

If a string is returned, the string should be interpreted as the name of a global variable. It should be the object’s local name relative to its module; the pickle module searches the module namespace to determine the object’s module. This behaviour is typically useful for singletons.

When a tuple is returned, it must be between two and five items long. Optional items can either be omitted, or None can be provided as their value. The semantics of each item are in order:

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.
- Optionally, the object’s state, which will be passed to the object’s __setstate__() method as previously described. If the object has no such method then, the value must be a dictionary and it will be added to the object’s __dict__ attribute.
- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using obj.append(item) or, in batch, using obj.extend(list_of_items). This is primarily used for list subclasses, but may be used by other classes as long as they have append() and extend() methods with the
appropriate signature. (Whether append() or extend() is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

- Optionally, an iterator (not a sequence) yielding successive key-value pairs. These items will be stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`. 

    `object.__reduce_ex__(protocol)`

    Alternatively, a `__reduce_ex__()` method may be defined. The only difference is this method should take a single integer argument, the protocol version. When defined, pickle will prefer it over the `__reduce__()` method. In addition, `__reduce__()` automatically becomes a synonym for the extended version. The main use for this method is to provide backwards-compatible reduce values for older Python releases.

### Persistence of External Objects

For the benefit of object persistence, the `pickle` module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a persistent ID, which should be either a string of alphanumeric characters (for protocol 0) [5] or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the `pickle` module; it will delegate this resolution to the user defined methods on the pickler and unpickler, `persistent_id()` and `persistent_load()` respectively.
To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom `persistent_load()` method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```python
# Simple example presenting how persistent ID can be used to pickle external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):
    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple containing a tag and key, which refers to a specific record.
            return ("MemoRecord", obj.key)
        else:
```
class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database.
            cursor.execute("SELECT * FROM memos WHERE key=?", (key_id,))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referred
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")

def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:"
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)"
    tasks = (..."}
'give food to fish',
'prepare group meeting',
'fight with a zebra',
)
for task in tasks:
cursor.execute("INSERT INTO memos VALUES(NULL,?)"

# Fetch the records to be pickled.
cursor.execute("SELECT * FROM memos")
memos = [MemoRecord(key, task) for key, task in
# Save the records using our custom DBPickler.
file = io.BytesIO()
DBPickler(file).dump(memos)

print("Pickled records:")
pprint.pprint(memos)

# Update a record, just for good measure.
cursor.execute("UPDATE memos SET task='learn ita"

# Load the records from the pickle data stream.
file.seek(0)
memos = DBUnpickler(file, conn).load()

print("Unpickled records:")
pprint.pprint(memos)

if __name__ == '__main__':
  main()

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a
pickler with a private dispatch table.

The global dispatch table managed by the `copyreg` module is available as `copyreg.dispatch_table`. Therefore, one may choose to use a modified copy of `copyreg.dispatch_table` as a private dispatch table.

For example

```python
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

creates an instance of `pickle.Pickler` with a private dispatch table which handles the `SomeClass` class specially. Alternatively, the code

```python
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same, but all instances of `MyPickler` will by default share the same dispatch table. The equivalent code using the `copyreg` module is

```python
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

Handling Stateful Objects
Here's an example that shows how to modify pickling behavior for a class. The TextReader class opens a text file, and returns the line number and line contents each time its readline() method is called. If a TextReader instance is pickled, all attributes except the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The __setstate__() and __getstate__() methods are used to implement this behavior.

class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('
'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__.  
        # all our instance attributes. Always use the 
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpickable entries.
        del state['file']
        return state
def __setstate__(self, state):
    # Restore instance attributes (i.e., filename and lineno).
    self.__dict__.update(state)
    # Restore the previously opened file's state.
    # Reopen it and read from it until the line count is restored.
    file = open(self.filename)
    for _ in range(self.lineno):
        file.readline()
    # Finally, save the file.
    self.file = file

A sample usage might be something like this:

```python
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```
Restricting Globals

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded:

```python
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'
hello world
0
```

In this example, the unpickler imports the `os.system()` function and then apply the string argument “echo hello world”. Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing `Unpickler.find_class()`. Unlike its name suggests, `Unpickler.find_class()` is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the `builtins` module to be loaded:

```python
import builtins
import io
import pickle

safe_builtins = {
    'range',
```
class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s'
        (module, name))

    def restricted_loads(s):
        """Helper function analogous to pickle.loads().""
        return RestrictedUnpickler(io.BytesIO(s)).load()
As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in `xmlrpc.client` or third-party solutions.
Performance

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the pickle module has a transparent optimizer written in C.
Examples

For the simplest code, use the \texttt{dump()} and \texttt{load()} functions.

\begin{verbatim}
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ('character string', b'byte string'),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)

The following example reads the resulting pickled data.

\begin{verbatim}
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically,
    # have to specify it.
data = pickle.load(f)
\end{verbatim}

See also:

Module \texttt{copyreg}

Pickle interface constructor registration for extension types.

Module \texttt{pickletools}

Tools for working with and analyzing pickled data.
Module **shelve**
Indexed databases of objects; uses **pickle**.

Module **copy**
Shallow and deep object copying.

Module **marshal**
High-performance serialization of built-in types.

Footnotes

[1] Don’t confuse this with the **marshal** module

[2] This is why **lambda** functions cannot be pickled: all **lambda** functions share the same name: `<lambda>`.

[3] The exception raised will likely be an **ImportError** or an **AttributeError** but it could be something else.

[4] The **copy** module uses this protocol for shallow and deep copying operations.

[5] The limitation on alphanumerical characters is due to the fact the persistent IDs, in protocol 0, are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickle will become unreadable.
copyreg — Register pickle support functions

Source code: Lib/copyreg.py

The copyreg module offers a way to define functions used while pickling specific objects. The pickle and copy modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

**copyreg.**constructor**(object)**

Declares object to be a valid constructor. If object is not callable (and hence not valid as a constructor), raises TypeError.

**copyreg.**pickle**(type, function, constructor=None)**

Declares that function should be used as a “reduction” function for objects of type type. function should return either a string or a tuple containing two or three elements.

The optional constructor parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by function at pickling time. TypeError will be raised if object is a class or constructor is not callable.

See the pickle module for more details on the interface expected of function and constructor. Note that the dispatch_table
attribute of a pickler object or subclass of `pickle.Pickler` can also be used for declaring reduction functions.
Example

The example below would like to show how to register a pickle function and how it will be used:

```python
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
... >>>
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
... >>>
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)  # doctest: +SKIP
pickling a C instance...
>>> p = pickle.dumps(c)  # doctest: +SKIP
pickling a C instance...
```
**shelve** — Python object persistence

Source code: Lib/shelve.py

A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

```python
shelve.open(filename, flag='c', protocol=None, writeback=False)
```

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional flag parameter has the same interpretation as the flag parameter of `dbm.open()`.

By default, version 3 pickles are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see Example). If the optional writeback parameter is set to `True`, all entries accessed are also cached in memory, and written back on `sync()`.
and `close()`, this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

**Note:** Do not rely on the shelf being closed automatically; always call `close()` explicitly when you don’t need it any more, or use `shelve.open()` as a context manager:

```python
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

**Warning:** Because the `shelve` module is backed by `pickle`, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support all methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

**Shelf.** `sync()`

Write back all entries in the cache if the shelf was opened with `writeback` set to `True`. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with `close()`.

**Shelf.** `close()`

Synchronize and close the persistent `dict` object. Operations on a
closed shelf will fail with a `ValueError`.

See also: Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.
Restrictions

- The choice of which database package will be used (such as `dbm.ndbm` or `dbm.gnu`) depends on which interface is available. Therefore it is not safe to open the database directly using `dbm`. The database is also (unfortunately) subject to the limitations of `dbm`, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

- The `shelve` module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

```python
class shelve.Shelf(dict, protocol=None, writeback=False, keyencoding='utf-8')
```

A subclass of `collections.abc.MutableMapping` which stores pickled values in the `dict` object.

By default, version 3 pickles are used to serialize values. The version of the pickle protocol can be specified with the `protocol` parameter. See the `pickle` documentation for a discussion of the pickle protocols.

If the `writeback` parameter is `True`, the object will hold a cache of all entries accessed and write them back to the `dict` at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a
long time.

The keyencoding parameter is the encoding used to encode keys before they are used with the underlying dict.

A Shelf object can also be used as a context manager, in which case it will be automatically closed when the with block ends.

Changed in version 3.2: Added the keyencoding parameter; previously, keys were always encoded in UTF-8.

Changed in version 3.4: Added context manager support.

class shelve.BsdDbShelf(dict, protocol=None, writeback=False, keyencoding='utf-8')
A subclass of Shelf which exposes first(), next(), previous(), last() and set_location() which are available in the third-party bsddb module from pybsd but not in other database modules. The dict object passed to the constructor must support those methods. This is generally accomplished by calling one of bsddb.hashopen(), bsddb.btopen() or bsddb.rnopen(). The optional protocol, writeback, and keyencoding parameters have the same interpretation as for the Shelf class.

class shelve.DbfilenameShelf(filename, flag='c', protocol=None, writeback=False)
A subclass of Shelf which accepts a filename instead of a dict-like object. The underlying file will be opened using dbm.open(). By default, the file will be created and opened for both read and write. The optional flag parameter has the same interpretation as for the open() function. The optional protocol and writeback
parameters have the same interpretation as for the `Shelf` class.
Example

To summarize the interface (key is a string, data is an arbitrary object):

```python
import shelve

d = shelve.open(filename)  # open -- file may get suffixed by low-level library

d[key] = data  # store data at key (overusing an existing key)
data = d[key]  # retrieve a COPY of data  # if no such key)
del d[key]  # delete data stored at key  # if no such key)

flag = key in d  # true if the key exists
klist = list(d.keys())  # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]  # this works as expected,
d['xx'].append(3)  # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to
temp = d['xx']  # extracts the copy
temp.append(5)  # mutates the copy
d['xx'] = temp  # stores the copy right back

# or, d=shelve.open(filename,writeback=True) would let
# d['xx'].append(5) and have it work as expected, but
# consume more memory and make the d.close() operation

d.close()  # close it
```
See also:

**Module dbm**
Generic interface to `dbm`-style databases.

**Module pickle**
Object serialization used by `shelve`. 
This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does). [1]

This is not a general “persistence” module. For general persistence and transfer of Python objects through RPC calls, see the modules pickle and shelve. The marshal module exists mainly to support reading and writing the “pseudo-compiled” code for Python modules of .pyc files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. If you’re serializing and de-serializing Python objects, use the pickle module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

Warning: The marshal module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: booleans, integers, floating point numbers, complex numbers, strings,
bytes, bytearrays, tuples, lists, sets, frozensets, dictionaries, and code objects, where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported. The singletons None, Ellipsis and StopIteration can also be marshalled and unmarshalled. For format version lower than 3, recursive lists, sets and dictionaries cannot be written (see below).

There are functions that read/write files as well as functions operating on bytes-like objects.

The module defines these functions:

```
marshal.dump(value, file[, version])
```

Write the value on the open file. The value must be a supported type. The file must be a writeable binary file.

If the value has (or contains an object that has) an unsupported type, a ValueError exception is raised — but garbage data will also be written to the file. The object will not be properly read back by load().

The version argument indicates the data format that dump should use (see below).

```
marshal.load(file)
```

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version’s incompatible marshal format), raise EOFError, ValueError or TypeError. The file must be a readable binary file.

**Note:** If an object containing an unsupported type was
marshalled with `dump()`, `load()` will substitute None for the unmarshallable type.

`marshal.dumps(value[, version])`
Return the bytes object that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a `ValueError` exception if value has (or contains an object that has) an unsupported type.

The `version` argument indicates the data format that `dumps` should use (see below).

`marshal.loads(bytes)`
Convert the bytes-like object to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Extra bytes in the input are ignored.

In addition, the following constants are defined:

`marshal.version`
Indicates the format that the module uses. Version 0 is the historical format, version 1 shares interned strings and version 2 uses a binary format for floating point numbers. Version 3 adds support for object instancing and recursion. The current version is 4.

**Footnotes**

The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term “marshalling” for shipping of data around in a self-contained form. Strictly speaking, “to marshal” means to convert some data from internal to external form (in an RPC
buffer for instance) and “unmarshalling” for the reverse process.
dbm — Interfaces to Unix “databases”

Source code: Lib/dbm/__init__.py

dbm is a generic interface to variants of the DBM database — dbm.gnu or dbm.ndbm. If none of these modules is installed, the slow-but-simple implementation in module dbm.dumb will be used. There is a third party interface to the Oracle Berkeley DB.

exception dbm.error

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named dbm.error as the first item — the latter is used when dbm.error is raised.

dbm.whichdb(filename)

This function attempts to guess which of the several simple database modules available — dbm.gnu, dbm.ndbm or dbm.dumb — should be used to open a given file.

Returns one of the following values: None if the file can’t be opened because it’s unreadable or doesn’t exist; the empty string (‘’) if the file’s format can’t be guessed; or a string containing the required module name, such as 'dbm.ndbm' or 'dbm.gnu'.

dbm.open(file, flag='r', mode=0o666)

Open the database file file and return a corresponding object.
If the database file already exists, the `whichdb()` function is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional *flag* argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn't exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

The object returned by `open()` supports the same basic functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()`.

*Changed in version 3.2:* `get()` and `setdefault()` are now available in all database modules.

Key and values are always stored as bytes. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will
automatically close them when done.

*Changed in version 3.4:* Added native support for the context management protocol to the objects returned by `open()`.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```python
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db['hello'] = 'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db[b'www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most likely a TypeError).
    db['www.yahoo.com'] = 4

    # db is automatically closed when leaving the with statement.
```

**See also:**

*Module shelve*
Persistence module which stores non-string data.

The individual submodules are described in the following sections.
**dbm.gnu** — GNU’s reimplementation of dbm

**Source code:** Lib/dbm/gnu.py

This module is quite similar to the `dbm` module, but uses the GNU library `gdbm` instead to provide some additional functionality. Please note that the file formats created by `dbm.gnu` and `dbm.ndbm` are incompatible.

The `dbm.gnu` module provides an interface to the GNU DBM library. `dbm.gnu.gdbm` objects behave like mappings (dictionaries), except that keys and values are always converted to bytes before storing. Printing a `gdbm` object doesn't print the keys and values, and the `items()` and `values()` methods are not supported.

`exception dbm.gnu.error`

Raised on `dbm.gnu`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

`dbm.gnu.open(filename[, flag[, mode]])`

Open a `gdbm` database and return a `gdbm` object. The `filename` argument is the name of the database file.

The optional `flag` argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
</tbody>
</table>
Open database for reading and writing, creating it if it doesn’t exist

Always create a new, empty database, open for reading and writing

The following additional characters may be appended to the flag to control how the database is opened:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
<tr>
<td>'f'</td>
<td>Open the database in fast mode. Writes to the database will not be synchronized.</td>
</tr>
<tr>
<td>'s'</td>
<td>Synchronized mode. This will cause changes to the database to be immediately written to the file.</td>
</tr>
<tr>
<td>'u'</td>
<td>Do not lock database.</td>
</tr>
</tbody>
</table>

Not all flags are valid for all versions of gdbm. The module constant open_flags is a string of supported flag characters. The exception error is raised if an invalid flag is specified.

The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666.

In addition to the dictionary-like methods, gdbm objects have the following methods:

**gdbm.firstkey()**

It’s possible to loop over every key in the database using this method and the nextkey() method. The traversal is ordered by gdbm’s internal hash values, and won’t be sorted by the key values. This method returns the starting key.

**gdbm.nextkey(key)**
Returns the key that follows key in the traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:

```python
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

gdbm. reorganize()

If you have carried out a lot of deletions and would like to shrink the space used by the gdbm file, this routine will reorganize the database. gdbm objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm. sync()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm. close()

Close the gdbm database.
The `dbm.ndbm` module provides an interface to the Unix “(n)dbm” library. Dbm objects behave like mappings (dictionaries), except that keys and values are always stored as bytes. Printing a `dbm` object doesn’t print the keys and values, and the `items()` and `values()` methods are not supported.

This module can be used with the “classic” ndbm interface or the GNU GDBM compatibility interface. On Unix, the `configure` script will attempt to locate the appropriate header file to simplify building this module.

**exception** `dbm.ndbm.error`

Raised on `dbm.ndbm`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

**dbm.ndbm.library**

Name of the `ndbm` implementation library used.

**dbm.ndbm.open(filename[, flag[, mode]])**

Open a dbm database and return a `ndbm` object. The `filename` argument is the name of the database file (without the `.dir` or `.pag` extensions).

The optional `flag` argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn't exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal \texttt{0o666} (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, \texttt{ndbm} objects provide the following method:

\texttt{ndbm.close()}  
Close the \texttt{ndbm} database.
The `dbm.dumb` module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as `dbm.gnu` no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

The module defines the following:

- **exception** `dbm.dumb.error`  
  Raised on `dbm.dumb`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

- **`dbm.dumb.open(filename[, flag[, mode]])`**  
  Open a dumbdbm database and return a dumbdbm object. The `filename` argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with `.dat` and `.dir` extensions are created.

  The optional `flag` argument supports only the semantics of 'c' and 'n' values. Other values will default to database being always
opened for update, and will be created if it does not exist.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

**Warning:** It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python’s AST compiler.

*Changed in version 3.5:* `open()` always creates a new database when the flag has the value `'n'`.  

*Deprecated since version 3.6, will be removed in version 3.8:* Creating database in `'r'` and `'w'` modes. Modifying database in `'r'` mode.

In addition to the methods provided by the `collections.abc.MutableMapping` class, `dumbdbm` objects provide the following methods:

**dumbdbm**. `sync()`  
Synchronize the on-disk directory and data files. This method is called by the `Shelve.sync()` method.

**dumbdbm**. `close()`  
Close the `dumbdbm` database.
sqlite3 — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It’s also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```python
import sqlite3
cconn = sqlite3.connect('example.db')
```

You can also supply the special name :memory: to create a database in RAM.

Once you have a Connection, you can create a Cursor object and call its `execute()` method to perform SQL commands:

```python
c = conn.cursor()
```
# Create table
```python
c.execute('''CREATE TABLE stocks
    (date text, trans text, symbol text, qty real, price real)''')
```

# Insert a row of data
```python
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)"
```

# Save (commit) the changes
```python
conn.commit()
```

# We can also close the connection if we are done with it.
```python
conn.close()
```

The data you've saved is persistent and is available in subsequent sessions:

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see [https://xkcd.com/327/](https://xkcd.com/327/) for humorous example of what can go wrong).

Instead, use the DB-API's parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'"

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
             ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)'

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's fetchone() method to retrieve a single matching row, or call fetchall() to get a list of the matching rows.

This example uses the iterator form:

```python
for row in c.execute('SELECT * FROM stocks ORDER BY price')
    print(row)
```

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)

See also:
https://github.com/ghaering/pysqlite
The pysqlite web page – sqlite3 is developed externally under the name “pysqlite”.

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

https://www.w3schools.com/sql/
Tutorial, reference and examples for learning SQL syntax.

PEP 249 - Database API Specification 2.0
PEP written by Marc-André Lemburg.
Module functions and constants

```python
sqlite3.version
   The version number of this module, as a string. This is not the version of the SQLite library.
```

```python
sqlite3.version_info
   The version number of this module, as a tuple of integers. This is not the version of the SQLite library.
```

```python
sqlite3.sqlite_version
   The version number of the run-time SQLite library, as a string.
```

```python
sqlite3.sqlite_version_info
   The version number of the run-time SQLite library, as a tuple of integers.
```

```python
sqlite3.PARSE_DECLTYPES
   This constant is meant to be used with the detect_types parameter of the connect() function.

   Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i.e. for “integer primary key”, it will parse out “integer”, or for “number(10)” it will parse out “number”. Then for that column, it will look into the converters dictionary and use the converter function registered for that type there.
```

```python
sqlite3.PARSE_COLNAMES
   This constant is meant to be used with the detect_types parameter of the connect() function.
```
Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that ‘mytype’ is the type of the column. It will try to find an entry of ‘mytype’ in the converters dictionary and then use the converter function found there to return the value. The column name found in `Cursor.description` is only the first word of the column name, i. e. if you use something like 'as "x [datetime]"' in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be “x”.

```
sqlite3.connect(database[, timeout, detect_types,
        isolation_level, check_same_thread, factory, cached_statements, uri])
```

Opens a connection to the SQLite database file `database`. By default returns a `Connection` object, unless a custom `factory` is given.

`database` is a path-like object giving the pathname (absolute or relative to the current working directory) of the database file to be opened. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The `timeout` parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the `isolation_level` parameter, please see the `isolation_level` property of `Connection` objects.
SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The `detect_types` parameter and the using custom `converters` registered with the module-level `register_converter()` function allow you to easily do that.

`detect_types` defaults to 0 (i.e. off, no type detection), you can set it to any combination of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to turn type detection on.

By default, `check_same_thread` is `True` and only the creating thread may use the connection. If set `False`, the returned connection may be shared across multiple threads. When using multiple threads with the same connection writing operations should be serialized by the user to avoid data corruption.

By default, the `sqlite3` module uses its `Connection` class for the connect call. You can, however, subclass the `Connection` class and make `connect()` use your class instead by providing your class for the `factory` parameter.

Consult the section `SQLite and Python types` of this manual for details.

The `sqlite3` module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the `cached_statements` parameter. The currently implemented default is to cache 100 statements.

If `uri` is true, `database` is interpreted as a URI. This allows you to specify options. For example, to open a database in read-only mode you can use:
```
import sqlite3

db = sqlite3.connect('file:path/to/database?mode=ro')
```

More information about this feature, including a list of recognized options, can be found in the SQLite URI documentation.

*Changed in version 3.4:* Added the `uri` parameter.

*Changed in version 3.7:* `database` can now also be a path-like object, not only a string.

**`sqlite3.register_converter()`**

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type `typename`. Confer the parameter `detect_types` of the `connect()` function for how the type detection works. Note that `typename` and the name of the type in your query are matched in case-insensitive manner.

**`sqlite3.register_adapter()`**

Registers a callable to convert the custom Python type `type` into one of SQLite's supported types. The callable `callable` accepts a single parameter the Python value, and must return a value of the following types: `int`, `float`, `str` or `bytes`.

**`sqlite3.complete_statement()`**

Returns `True` if the string `sql` contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, as in the following example:
# A minimal SQLite shell for experiments

```
import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.

print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)
            if buffer.lstrip().upper().startswith('PRINT')
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
            buffer = ""

con.close()
```

`sqlite3.enable_callback_tracebacks(flag)`

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with `flag` set to `True`. 
Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.
Connection Objects

`class sqlite3.Connection`

A SQLite database connection has the following attributes and methods:

**isolation_level**
Get or set the current default isolation level. `None` for autocommit mode or one of “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”. See section Controlling Transactions for a more detailed explanation.

**in_transaction**
`True` if a transaction is active (there are uncommitted changes), `False` otherwise. Read-only attribute.

*New in version 3.2.*

**cursor(factory=Cursor)**
The cursor method accepts a single optional parameter `factory`. If supplied, this must be a callable returning an instance of `Cursor` or its subclasses.

**commit()**
This method commits the current transaction. If you don’t call this method, anything you did since the last call to `commit()` is not visible from other database connections. If you wonder why you don’t see the data you’ve written to the database, please check you didn’t forget to call this method.

**rollback()**
This method rolls back any changes to the database since the
last call to `commit()`.

**close()**
This closes the database connection. Note that this does not automatically call `commit()`. If you just close your database connection without calling `commit()` first, your changes will be lost!

**execute**(sql[, parameters])
This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor’s `execute()` method with the `parameters` given, and returns the cursor.

**executemany**(sql[, parameters])
This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor’s `executemany()` method with the `parameters` given, and returns the cursor.

**executescript**(sql_script)
This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor’s `executescript()` method with the given `sql_script`, and returns the cursor.

**create_function**(name, num_params, func)
Creates a user-defined function that you can later use from within SQL statements under the function name `name`. `num_params` is the number of parameters the function accepts (if `num_params` is -1, the function may take any number of arguments), and `func` is a Python callable that is called as the SQL function.
The function can return any of the types supported by SQLite: bytes, str, int, float and None.

Example:

```python
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(':memory:)
con.create_function('md5', 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b'foo',))
print(cur.fetchone()[0])
```

`create_aggregate(name, num_params, aggregate_class)`

Creates a user-defined aggregate function.

The aggregate class must implement a `step` method, which accepts the number of parameters `num_params` (if `num_params` is -1, the function may take any number of arguments), and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite: bytes, str, int, float and None.

Example:

```python
import sqlite3

class MySum:
    def __init__(self):
```
```python
self.count = 0

def step(self, value):
    self.count += value

def finalize(self):
    return self.count

con = sqlite3.connect(':memory:')
con.create_aggregate('mysum', 1, MySum)
cur = con.cursor()
cur.execute('create table test(i)')
cur.execute('insert into test(i) values (1)')
cur.execute('insert into test(i) values (2)')
cur.execute('select mysum(i) from test')
print(cur.fetchone()[0])
```

### `create_collation(name, callable)`

Creates a collation with the specified `name` and `callable`. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don’t affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts “the wrong way”:

```python
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        ...  # implementation
```
```python
return 0
elif string1 < string2:
    return 1
else:
    return -1
```

```python
con = sqlite3.connect(":memory:"
con.create_collation("reverse", collate_reverse)
cur = con.cursor()
cur.execute("create table test(x)"
cur.executemany("insert into test(x) values (?)")
cur.execute("select x from test order by x collate reverse"
for row in cur:
    print(row)
con.close()
```

To remove a collation, call `create_collation` with `None` as callable:

```python
con.create_collation("reverse", None)
```

**interrupt**

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

**set_authorizer**(authorizer_callback)

This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return SQLITE_OK if access is allowed, SQLITE_DENY if the entire SQL statement should be aborted with an error and SQLITE_IGNORE if the column should be treated as a NULL value. These constants are available in the
The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database (“main”, “temp”, etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

**set_progress_handler** *(handler, n)*

This routine registers a callback. The callback is invoked for every `n` instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for `handler`.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise an `OperationalError` exception.

**set_trace_callback** *(trace_callback)*

Registers `trace_callback` to be called for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as
string) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the `Cursor.execute()` methods. Other sources include the transaction management of the Python module and the execution of triggers defined in the current database.

Passing `None` as `trace_callback` will disable the trace callback.

*New in version 3.3.*

**enable_load_extension**(`enabled`)  
This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See [1].

*New in version 3.2.*

```python
import sqlite3

con = sqlite3.connect(":memory:"

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using
# con.load_extension("./fts3.so")

# disable extension loading again
```
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)"
con.executescript(""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sugar flour butter');
"")
for row in con.execute("select rowid, name, ingredients from recipe where name match 'pie'"
print(row)

load_extension(path)
This routine loads a SQLite extension from a shared library. You have to enable extension loading with enable_load_extension() before you can use this routine.

Loadable extensions are disabled by default. See [1].

New in version 3.2.

row_factory
You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Example:

import sqlite3
def dict_factory(cursor, row):

If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting `row_factory` to the highly-optimized `sqlite3.Row` type. `Row` provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a db_row based solution.

**text_factory**

Using this attribute you can control what objects are returned for the TEXT data type. By default, this attribute is set to `str` and the `sqlite3` module will return Unicode objects for TEXT. If you want to return bytestrings instead, you can set it to `bytes`.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```python
import sqlite3

con = sqlite3.connect(":memory:")
```
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory.
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8")
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"

**total_changes**

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

**iterdump()**

Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the .dump command in the sqlite3 shell.
Example:

```python
# Convert file existing_db.db to SQL dump file
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%%s
' % line)
```

```
backup(target, *, pages=0, progress=None, name="main", sleep=0.250)
```

This method makes a backup of a SQLite database even while it's being accessed by other clients, or concurrently by the same connection. The copy will be written into the mandatory argument `target`, that must be another `Connection` instance.

By default, or when `pages` is either 0 or a negative integer, the entire database is copied in a single step; otherwise the method performs a loop copying up to `pages` pages at a time.

If `progress` is specified, it must either be `None` or a callable object that will be executed at each iteration with three integer arguments, respectively the `status` of the last iteration, the `remaining` number of pages still to be copied and the `total` number of pages.

The `name` argument specifies the database name that will be copied: it must be a string containing either "main", the default, to indicate the main database, "temp" to indicate the temporary database or the name specified after the AS keyword in an `ATTACH DATABASE` statement for an attached database.
The sleep argument specifies the number of seconds to sleep by between successive attempts to backup remaining pages, can be specified either as an integer or a floating point value.

Example 1, copy an existing database into another:

```python
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total}
con = sqlite3.connect('existing_db.db')
with sqlite3.connect('backup.db') as bck:
    con.backup(bck, pages=1, progress=progress)
```

Example 2, copy an existing database into a transient copy:

```python
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

Availability: SQLite 3.6.11 or higher

*New in version 3.7.*
Cursor Objects

class sqlite3.Cursor

A Cursor instance has the following attributes and methods.

execute(sql[, parameters])

Executes an SQL statement. The SQL statement may be parameterized (i.e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

Here's an example of both styles:

```python
import sqlite3

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table people (name_last, age)"

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)"

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age"

print(cur.fetchone())
```

execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use executescript() if you want to execute
multiple SQL statements with one call.

**executemany**(sql, seq_of_parameters)

Executes an SQL command against all parameter sequences or mappings found in the sequence seq_of_parameters. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```python
import sqlite3
class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)"

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)")
cur.execute("select c from characters")
print(cur.fetchall())
```

Here's a shorter example using a generator:
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)"

cur.executemany("insert into characters(c) values (?)"

cur.execute("select c from characters"
print(cur.fetchall())

executescript(sql_script)
This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

sql_script can be an instance of str.

Example:

import sqlite3

con = sqlite3.connect(':memory:"
cur = con.cursor()
cur.executescript(""
    create table person(
        firstname,
        lastname,
        age
    )
);
create table book(
    title,
    author,
    published
);

insert into book(title, author, published)
values ('Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987);

fetchone()

Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

fetchmany(size=cursor.arraysize)

Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the size parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one fetchmany()
call to the next.

**fetchall()**
Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

**close()**
Close the cursor now (rather than whenever __del__ is called).

The cursor will be unusable from this point forward; a **ProgrammingError** exception will be raised if any operation is attempted with the cursor.

**rowcount**
Although the **Cursor** class of the sqlite3 module implements this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky.

For **executemany()** statements, the number of modifications are summed up into **rowcount**.

As required by the Python DB API Spec, the **rowcount** attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface". This includes **SELECT** statements because we cannot determine the number of rows a query produced until all rows were fetched.

With SQLite versions before 3.6.5, **rowcount** is set to 0 if you make a DELETE FROM table without any condition.
lastrowid

This read-only attribute provides the rowid of the last modified row. It is only set if you issued an `INSERT` or a `REPLACE` statement using the `execute()` method. For operations other than `INSERT` or `REPLACE` or when `executemany()` is called, `lastrowid` is set to `None`.

If the `INSERT` or `REPLACE` statement failed to insert the previous successful rowid is returned.

*Changed in version 3.6:* Added support for the `REPLACE` statement.

arraysize

Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

description

This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

collection

This read-only attribute provides the SQLite database `Connection` used by the `Cursor` object. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to `con`: 
```python
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```
Row Objects

class sqlite3.Row

A Row instance serves as a highly optimized row_factory for Connection objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and len().

If two Row objects have exactly the same columns and their members are equal, they compare equal.

keys()

This method returns a list of column names. Immediately after a query, it is the first member of each tuple in Cursor.description.

Changed in version 3.5: Added support of slicing.

Let's assume we initialize a table as in the example given above:

```python
conn = sqlite3.connect(':memory:)
c = conn.cursor()
c.execute('''create table stocks
date text, trans text, symbol text,
qty real, price real)'''
)'
c.execute('''insert into stocks
values ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)"
)'
conn.commit()
c.close()
```

Now we plug Row in:
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
Exceptions

`Exception sqlite3.Warning`
A subclass of `Exception`.

`Exception sqlite3.Error`
The base class of the other exceptions in this module. It is a subclass of `Exception`.

`Exception sqlite3.DatabaseError`
Exception raised for errors that are related to the database.

`Exception sqlite3.IntegrityError`
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of `DatabaseError`.

`Exception sqlite3.ProgrammingError`
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of `DatabaseError`.

`Exception sqlite3.OperationalError`
Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, etc. It is a subclass of `DatabaseError`.

`Exception sqlite3.NotSupportedError`
Exception raised in case a method or database API was used which is not supported by the database, e.g. calling the
rollback() method on a connection that does not support transaction or has transactions turned off. It is a subclass of DatabaseError.
SQLite and Python types

Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

<table>
<thead>
<tr>
<th>Python type</th>
<th>SQLite type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NULL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>str</td>
<td>TEXT</td>
</tr>
<tr>
<td>bytes</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

This is how SQLite types are converted to Python types by default:

<table>
<thead>
<tr>
<th>SQLite type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>None</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>TEXT</td>
<td>depends on text_factory, str by default</td>
</tr>
<tr>
<td>BLOB</td>
<td>bytes</td>
</tr>
</tbody>
</table>

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in a SQLite database via object
adaptation, and you can let the sqlite3 module convert SQLite types to different Python types via converters.

**Using adapters to store additional Python types in SQLite databases**

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must adapt them to one of the sqlite3 module’s supported types for SQLite: one of NoneType, int, float, str, bytes.

There are two ways to enable the sqlite3 module to adapt a custom Python type to one of the supported ones.

**Letting your object adapt itself**

This is a good approach if you write the class yourself. Let’s suppose you have a class like this:

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you’ll have to choose one of the supported types first to be used for representing the point. Let’s just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter `protocol` will be `PrepareProtocol`.

```python
import sqlite3
```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%.f;%.f" % (self.x, self.y)

con = sqlite3.connect(":memory:"))
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with register_adapter().

import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def adapt_point(point):
        return "%.f;%.f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:"))
cur = con.cursor()

p = Point(4.0, -3.2)
The `sqlite3` module has two default adapters for Python's built-in `datetime.date` and `datetime.datetime` types. Now let's suppose we want to store `datetime.datetime` objects not in ISO representation, but as a Unix timestamp.

```python
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(':memory:)
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])
```

Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.
First, we'll define a converter function that accepts the string as a parameter and constructs a Point object from it.

**Note:** Converter functions **always** get called with a `bytes` object, no matter under which data type you sent the value to SQLite.

```python
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

Now you need to make the `sqlite3` module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section `Module functions and constants`, in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.

```python
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

    def adapt_point(point):
        return ("%f;%f" % (point.x, point.y)).encode('ascii')
```
```python
def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p_point)"

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p)"

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()```
Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name “date” for `datetime.date` and under the name “timestamp” for `datetime.datetime`.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```python
import sqlite3
import datetime

con = sqlite3.connect(':memory:', detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)"

today = datetime.date.today()
own = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)",
cur.execute("select d, ts from test"
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as ",
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
```
If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.
Controlling Transactions

The underlying sqlite3 library operates in autocommit mode by default, but the Python sqlite3 module by default does not.

Autocommit mode means that statements that modify the database take effect immediately. A BEGIN or SAVEPOINT statement disables autocommit mode, and a COMMIT, a ROLLBACK, or a RELEASE that ends the outermost transaction, turns autocommit mode back on.

The Python sqlite3 module by default issues a BEGIN statement implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE).

You can control which kind of BEGIN statements sqlite3 implicitly executes via the isolation_level parameter to the connect() call, or via the isolation_level property of connections. If you specify no isolation_level, a plain BEGIN is used, which is equivalent to specifying DEFERRED. Other possible values are IMMEDIATE and EXCLUSIVE.

You can disable the sqlite3 module’s implicit transaction management by setting isolation_level to None. This will leave the underlying sqlite3 library operating in autocommit mode. You can then completely control the transaction state by explicitly issuing BEGIN, ROLLBACK, SAVEPOINT, and RELEASE statements in your code.

Changed in version 3.6: sqlite3 used to implicitly commit an open transaction before DDL statements. This is no longer the case.
Using **sqlite3** efficiently

Using shortcut methods

Using the nonstandard `execute()`, `executemany()` and `executescript()` methods of the `Connection` object, your code can be written more concisely because you don't have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a **SELECT** statement and iterate over it directly using only a single call on the `Connection` object.

```python
import sqlite3

persons = [
    ('Hugo', 'Boss'),
    ('Calvin', 'Klein')
]

con = sqlite3.connect(':memory:)

# Create the table
con.execute("create table person(firstname, lastname)"

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)"

# Print the table contents
for row in con.execute("select firstname, lastname from person"
    print(row)

print("I just deleted", con.execute("delete from person")
```
Accessing columns by name instead of by index

One useful feature of the sqlite3 module is the built-in sqlite3.Row class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```python
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["Age"]
```

Using the connection as a context manager

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```python
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)"

# Successful, con.commit() is called automatically after this statement
```
with con:
    con.execute("insert into person(firstname) values (?)")

# con.rollback() is called after the with block finishes with an exception, the exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)")
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
Common issues

Multithreading

Older SQLite versions had issues with sharing connections between threads. That's why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the `interrupt()` method, which only makes sense to call from a different thread.

Footnotes

(1, 2) The sqlite3 module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass –enable-loadable-sqlite-extensions to configure.
Data Compression and Archiving

The modules described in this chapter support data compression with the zlib, gzip, bzip2 and lzma algorithms, and the creation of ZIP- and tar-format archives. See also Archiving operations provided by the shutil module.

- **zlib** — Compression compatible with gzip
- **gzip** — Support for gzip files
  - Examples of usage
- **bz2** — Support for bzip2 compression
  - (De)compression of files
  - Incremental (de)compression
  - One-shot (de)compression
- **lzma** — Compression using the LZMA algorithm
  - Reading and writing compressed files
  - Compressing and decompressing data in memory
  - Miscellaneous
  - Specifying custom filter chains
  - Examples
- **zipfile** — Work with ZIP archives
  - ZipFile Objects
  - PyZipFile Objects
  - ZipInfo Objects
  - Command-Line Interface
    - Command-line options
- **tarfile** — Read and write tar archive files
  - TarFile Objects
  - TarInfo Objects
  - Command-Line Interface
    - Command-line options
  - Examples
- Supported tar formats
- Unicode issues
**zlib — Compression compatible with gzip**

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at [http://www.zlib.net](http://www.zlib.net). There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib’s functions have many options and often need to be used in a particular order. This documentation doesn’t attempt to cover all of the permutations; consult the zlib manual at [http://www.zlib.net/manual.html](http://www.zlib.net/manual.html) for authoritative information.

For reading and writing `.gz` files see the gzip module.

The available exception and functions in this module are:

* **`zlib.error`**

  Exception raised on compression and decompression errors.

* **`zlib.adler32(data[, value])`**

  Computes an Adler-32 checksum of `data`. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) The result is an unsigned 32-bit integer. If `value` is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used. Passing in `value` allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used
for authentication or digital signatures. Since the algorithm is
designed for use as a checksum algorithm, it is not suitable for use
as a general hash algorithm.

*Changed in version 3.0:* Always returns an unsigned value. To
generate the same numeric value across all Python versions and
platforms, use `adler32(data) & 0xffffffff`.

**zlib.compress** *(data, level=-1)*

Compresses the bytes in `data`, returning a bytes object containing
compressed data. `level` is an integer from 0 to 9 or -1 controlling
the level of compression; 1 (Z_BEST_SPEED) is fastest and
produces the least compression, 9 (Z_BEST_COMPRESSION) is
slowest and produces the most. 0 (Z_NO_COMPRESSION) is no
compression. The default value is -1 (Z_DEFAULT_COMPRESSION).
Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6). Raises the `error` exception if any
error occurs.

*Changed in version 3.6:* `level` can now be used as a keyword
parameter.

**zlib.compressobj** *(level=-1, method=DEFLATED,
wbits=MAX_WBITS, memLevel=DEF_MEM_LEVEL,
strategy=Z_DEFAULT_STRATEGY[, zdict]*)

Returns a compression object, to be used for compressing data
streams that won’t fit into memory at once.

`level` is the compression level – an integer from 0 to 9 or -1. A
value of 1 (Z_BEST_SPEED) is fastest and produces the least
compression, while a value of 9 (Z_BEST_COMPRESSION) is
slowest and produces the most. 0 (Z_NO_COMPRESSION) is no compression. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

`method` is the compression algorithm. Currently, the only supported value is DEFLATED.

The `wbits` argument controls the size of the history buffer (or the “window size”) used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to 15 (MAX_WBITS):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of `wbits` as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic gzip header and trailing checksum in the output.

The `memLevel` argument controls the amount of memory used for the internal compression state. Valid values range from 1 to 9. Higher values use more memory, but are faster and produce smaller output.

`strategy` is used to tune the compression algorithm. Possible values are Z_DEFAULT_STRATEGY, Z_FILTERED, Z_HUFFMAN_ONLY, Z_RLE (zlib 1.2.0.1) and Z_FIXED (zlib
1.2.2.2).

`zdict` is a predefined compression dictionary. This is a sequence of bytes (such as a `bytes` object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary.

*Changed in version 3.3:* Added the `zdict` parameter and keyword argument support.

**`zlib.crc32(data[, value])`**

Computes a CRC (Cyclic Redundancy Check) checksum of `data`. The result is an unsigned 32-bit integer. If `value` is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used. Passing in `value` allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

*Changed in version 3.0:* Always returns an unsigned value. To generate the same numeric value across all Python versions and platforms, use `crc32(data) & 0xffffffff`.

**`zlib.decompress(data, wbits=MAX_WBITS, bufsize=DEF_BUF_SIZE)`**

Decompresses the bytes in `data`, returning a `bytes` object containing the uncompressed data. The `wbits` parameter depends on the format of `data`, and is discussed further below. If `bufsize` is given, it is used as the initial size of the output buffer. Raises the `error` exception if any error occurs.
The *wbits* parameter controls the size of the history buffer (or “window size”), and what header and trailer format is expected. It is similar to the parameter for `compressobj()`, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

*bufsize* is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to `malloc()`.

*Changed in version 3.6*: *wbits* and *bufsize* can be used as keyword arguments.
zlib.\texttt{decompressobj}(\texttt{wbits=}MAX\_\texttt{WBITS}[, \texttt{zdict}])

Returns a decompression object, to be used for decompressing data streams that won’t fit into memory at once.

The \texttt{wbits} parameter controls the size of the history buffer (or the “window size”), and what header and trailer format is expected. It has the same meaning as described for \texttt{decompress()}. The \texttt{zdict} parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

\textbf{Note:} If \texttt{zdict} is a mutable object (such as a \texttt{bytearray}), you must not modify its contents between the call to \texttt{decompressobj()} and the first call to the decompressor’s \texttt{decompress()} method.

\textit{Changed in version 3.3:} Added the \texttt{zdict} parameter.

Compression objects support the following methods:

\texttt{Compress.\texttt{compress}(data)}

Compress \texttt{data}, returning a bytes object containing compressed data for at least part of the data in \texttt{data}. This data should be concatenated to the output produced by any preceding calls to the \texttt{compress()} method. Some input may be kept in internal buffers for later processing.

\texttt{Compress.\texttt{flush}([mode])}

All pending input is processed, and a bytes object containing the remaining compressed output is returned. \texttt{mode} can be selected from the constants \texttt{Z\_NO\_FLUSH}, \texttt{Z\_PARTIAL\_FLUSH},
Z_SYNC_FLUSH, Z_FULL_FLUSH, Z_BLOCK (zlib 1.2.3.4), or Z_FINISH, defaulting to Z_FINISH. Except Z_FINISH, all constants allow compressing further bytestrings of data, while Z_FINISH finishes the compressed stream and prevents compressing any more data. After calling flush() with mode set to Z_FINISH, the compress() method cannot be called again; the only realistic action is to delete the object.

**Compress. copy()**

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

Decompression objects support the following methods and attributes:

**Decompress. unused_data**

A bytes object which contains any bytes past the end of the compressed data. That is, this remains b'' until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is b'', an empty bytes object.

**Decompress. unconsumed_tail**

A bytes object that contains any data that was not consumed by the last decompress() call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent decompress() method call in order to get correct output.

**Decompress. eof**

A boolean indicating whether the end of the compressed data stream has been reached.
This makes it possible to distinguish between a properly-formed compressed stream, and an incomplete or truncated one.

New in version 3.3.

Decompress. **decompress**(data, max_length=0)
Decompress *data*, returning a bytes object containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the *decompress()* method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute *unconsumed_tail*. This bytestring must be passed to a subsequent call to *decompress()* if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and *unconsumed_tail* is empty.

*Changed in version 3.6: max_length can be used as a keyword argument.*

Decompress. **flush**(length)
All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling *flush()*, the *decompress()* method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

Decompress. **copy**()
Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

Information about the version of the zlib library in use is available through the following constants:

**zlib.ZLIB_VERSION**

The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as `ZLIB_RUNTIME_VERSION`.

**zlib.ZLIB_RUNTIME_VERSION**

The version string of the zlib library actually loaded by the interpreter.

*New in version 3.3.*

See also:

- **Module gzip**
  - Reading and writing **gzip**-format files.
- **http://www.zlib.net**
  - The zlib library home page.
- **http://www.zlib.net/manual.html**
  - The zlib manual explains the semantics and usage of the library’s many functions.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
gzip — Support for gzip files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class, as well as the open(), compress() and decompress() convenience functions. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Note that additional file formats which can be decompressed by the gzip and gunzip programs, such as those produced by compress and pack, are not supported by this module.

The module defines the following items:

gzip.open(filename, mode='rb', compresslevel=9, encoding=None, errors=None, newline=None)

Open a gzip-compressed file in binary or text mode, returning a file object.

The filename argument can be an actual filename (a str or bytes object), or an existing file object to read from or write to.

The mode argument can be any of 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x' or 'xb' for binary mode, or 'rt', 'at', 'wt', or
'xt' for text mode. The default is 'rb'.

The compresslevel argument is an integer from 0 to 9, as for the GzipFile constructor.

For binary mode, this function is equivalent to the GzipFile constructor: GzipFile(filename, mode, compresslevel). In this case, the encoding, errors and newline arguments must not be provided.

For text mode, a GzipFile object is created, and wrapped in an io.TextIOWrapper instance with the specified encoding, error handling behavior, and line ending(s).

*Changed in version 3.3:* Added support for filename being a file object, support for text mode, and the encoding, errors and newline arguments.

*Changed in version 3.4:* Added support for the 'x', 'xb' and 'xt' modes.

*Changed in version 3.6:* Accepts a path-like object.

```python
class gzip.GzipFile(filename=None, mode=None, compresslevel=9, fileobj=None, mtime=None)
```

Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the truncate() method. At least one of fileobj and filename must be given a non-trivial value.

The new class instance is based on fileobj, which can be a regular file, an io.BytesIO object, or any other object which simulates a file. It defaults to None, in which case filename is opened to
provide a file object.

When `fileobj` is not `None`, the `filename` argument is only used to be included in the gzip file header, which may include the original filename of the uncompressed file. It defaults to the filename of `fileobj`, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The `mode` argument can be any of `'r', 'rb', 'a', 'ab', 'w', 'wb', 'x', or 'xb'`, depending on whether the file will be read or written. The default is the mode of `fileobj` if discernible; otherwise, the default is `'rb'`.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use `open()` (or wrap your `GzipFile` with an `io.TextIOWrapper`).

The `compresslevel` argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The `mtime` argument is an optional numeric timestamp to be written to the last modification time field in the stream when compressing. It should only be provided in compression mode. If omitted or `None`, the current time is used. See the `mtime` attribute for more details.

Calling a `GzipFile` object's `close()` method does not close `fileobj`, since you might wish to append more material after the compressed data. This also allows you to pass an `io.BytesIO` object opened for writing as `fileobj`, and retrieve the resulting
memory buffer using the io.BytesIO object's getvalue() method.

GzipFile supports the io.BufferedIOBase interface, including iteration and the with statement. Only the truncate() method isn't implemented.

GzipFile also provides the following method and attribute:

peek(n)
Read n uncompressed bytes without advancing the file position. At most one single read on the compressed stream is done to satisfy the call. The number of bytes returned may be more or less than requested.

Note: While calling peek() does not change the file position of the GzipFile, it may change the position of the underlying file object (e.g. if the GzipFile was constructed with the fileobj parameter).

New in version 3.2.

mtime
When decompressing, the value of the last modification time field in the most recently read header may be read from this attribute, as an integer. The initial value before reading any headers is None.

All gzip compressed streams are required to contain this timestamp field. Some programs, such as gunzip, make use of the timestamp. The format is the same as the return value of time.time() and the st_mtime attribute of the object returned by os.stat().
*Changed in version 3.1:* Support for the `with` statement was added, along with the `mtime` constructor argument and `mtime` attribute.

*Changed in version 3.2:* Support for zero-padded and unseekable files was added.

*Changed in version 3.3:* The `io.BufferedIOBase.read1()` method is now implemented.

*Changed in version 3.4:* Added support for the `'x'` and `'xb'` modes.

*Changed in version 3.5:* Added support for writing arbitrary bytes-like objects. The `read()` method now accepts an argument of `None`.

*Changed in version 3.6:* Accepts a path-like object.

**gzip.**`.compress(data, compresslevel=9)`

Compress the `data`, returning a `bytes` object containing the compressed data. `compresslevel` has the same meaning as in the `GzipFile` constructor above.

*New in version 3.2.*

**gzip.**`.decompress(data)`

Decompress the `data`, returning a `bytes` object containing the uncompressed data.

*New in version 3.2.*
Examples of usage

Example of how to read a compressed file:

```python
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```python
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```python
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Example of how to GZIP compress a binary string:

```python
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

See also:

Module zlib

The basic data compression module needed to support the gzip
file format.
bz2 — Support for bzip2 compression

Source code: Lib/bz2.py

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The bz2 module contains:

- The `open()` function and `BZ2File` class for reading and writing compressed files.
- The `BZ2Compressor` and `BZ2Decompressor` classes for incremental (de)compression.
- The `compress()` and `decompress()` functions for one-shot (de)compression.

All of the classes in this module may safely be accessed from multiple threads.
(De)compression of files

bz2. open(filename, mode='r', compresslevel=9, encoding=None, errors=None, newline=None)

Open a bzip2-compressed file in binary or text mode, returning a file object.

As with the constructor for BZ2File, the filename argument can be an actual filename (a str or bytes object), or an existing file object to read from or write to.

The mode argument can be any of 'r', 'rb', 'w', 'wb', 'x', 'xb', 'a' or 'ab' for binary mode, or 'rt', 'wt', 'xt', or 'at' for text mode. The default is 'rb'.

The compresslevel argument is an integer from 1 to 9, as for the BZ2File constructor.

For binary mode, this function is equivalent to the BZ2File constructor: BZ2File(filename, mode, compresslevel=compresslevel). In this case, the encoding, errors and newline arguments must not be provided.

For text mode, a BZ2File object is created, and wrapped in an io.TextIOWrapper instance with the specified encoding, error handling behavior, and line ending(s).

New in version 3.3.

Changed in version 3.4: The 'x' (exclusive creation) mode was added.
**Changed in version 3.6:** Accepts a *path-like object*.

```python
class bz2.BZ2File(filename, mode='r', buffering=None, compresslevel=9)
```

Open a bzip2-compressed file in binary mode.

If `filename` is a `str` or `bytes` object, open the named file directly. Otherwise, `filename` should be a `file object`, which will be used to read or write the compressed data.

The `mode` argument can be either `'r'` for reading (default), `'w'` for overwriting, `'x'` for exclusive creation, or `'a'` for appending. These can equivalently be given as `'rb'`, `'wb'`, `'xb'` and `'ab'` respectively.

If `filename` is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

The `buffering` argument is ignored. Its use is deprecated.

If `mode` is `'w'` or `'a'`, `compresslevel` can be a number between 1 and 9 specifying the level of compression: 1 produces the least compression, and 9 (default) produces the most compression.

If `mode` is `'r'`, the input file may be the concatenation of multiple compressed streams.

*BZ2File* provides all of the members specified by the `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

*BZ2File* also provides the following method:
peek([n])

Return buffered data without advancing the file position. At least one byte of data will be returned (unless at EOF). The exact number of bytes returned is unspecified.

**Note:** While calling `peek()` does not change the file position of the `BZ2File`, it may change the position of the underlying file object (e.g. if the `BZ2File` was constructed by passing a file object for `filename`).

*New in version 3.3.*

*Changed in version 3.1:* Support for the `with` statement was added.

*Changed in version 3.3:* The `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` and `readinto()` methods were added.

*Changed in version 3.3:* Support was added for `filename` being a `file object` instead of an actual filename.

*Changed in version 3.3:* The `'a'` (append) mode was added, along with support for reading multi-stream files.

*Changed in version 3.4:* The `'x'` (exclusive creation) mode was added.

*Changed in version 3.5:* The `read()` method now accepts an argument of `None`.

*Changed in version 3.6:* Accepts a `path-like object`. 
Incremental (de)compression

\texttt{class bz2.BZ2Compressor} \texttt{(compresslevel=9)}

Create a new compressor object. This object may be used to compress data incrementally. For one-shot compression, use the \texttt{compress()} function instead.

\textit{compresslevel}, if given, must be a number between 1 and 9. The default is 9.

\texttt{compress(data)}

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty byte string otherwise.

When you have finished providing data to the compressor, call the \texttt{flush()} method to finish the compression process.

\texttt{flush()}

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

\texttt{class bz2.BZ2Decompressor}

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot compression, use the \texttt{decompress()} function instead.

\textbf{Note:} This class does not transparently handle inputs containing multiple compressed streams, unlike \texttt{decompress()} and \texttt{BZ2File}. If you need to decompress a multi-stream input with
BZ2Decompressor, you must use a new decompressor for each stream.

`decompress(data, max_length=-1)`

Decompress `data` (a `bytes-like object`), returning uncompressed data as bytes. Some of `data` may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If `max_length` is nonnegative, returns at most `max_length` bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide `data` as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than `max_length` bytes, or because `max_length` was negative), the `needs_input` attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an `EOFError`. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

*Changed in version 3.5:* Added the `max_length` parameter.

`eof`

`True` if the end-of-stream marker has been reached.

*New in version 3.3.*

`unused_data`
Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be b''.

**needs_input**

False if the decompress() method can provide more decompressed data before requiring new uncompressed input.

*New in version 3.5.*
One-shot (de)compression

bz2.\texttt{compress}(\textit{data}, \textit{compresslevel}=9)
Compress \textit{data}.

\textit{compresslevel}, if given, must be a number between 1 and 9. The default is 9.

For incremental compression, use a \texttt{BZ2Compressor} instead.

bz2.\texttt{decompress}(\textit{data})
Decompress \textit{data}.

If \textit{data} is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a \texttt{BZ2Decompressor} instead.

\textit{Changed in version} 3.3: Support for multi-stream inputs was added.
lzma — Compression using the LZMA algorithm

New in version 3.3.

Source code: Lib/lzma.py

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the .xz and legacy .lzma file formats used by the xz utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the bz2 module. However, note that LZMAFile is not thread-safe, unlike bz2.BZ2File, so if you need to use a single LZMAFile instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.LZMAError

   This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.
Reading and writing compressed files

```python
lzma.open(filename, mode="rb", *, format=None, check=-1,
          preset=None, filters=None, encoding=None, errors=None,
          newline=None)
```

Open an LZMA-compressed file in binary or text mode, returning a file object.

The `filename` argument can be either an actual file name (given as a `str`, `bytes` or path-like object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The `mode` argument can be any of "r", "rb", "w", "wb", "x", "xb", "a" or "ab" for binary mode, or "rt", "wt", "xt", or "at" for text mode. The default is "rb".

When opening a file for reading, the `format` and `filters` arguments have the same meanings as for `LZMADecompressor`. In this case, the `check` and `preset` arguments should not be used.

When opening a file for writing, the `format`, `check`, `preset` and `filters` arguments have the same meanings as for `LZMACompressor`.

For binary mode, this function is equivalent to the `LZMAFile` constructor: `LZMAFile(filename, mode, ...)`. In this case, the `encoding`, `errors` and `newline` arguments must not be provided.

For text mode, a `LZMAFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s).

*Changed in version 3.4:* Added support for the "x", "xb" and
"xt" modes.

Changed in version 3.6: Accepts a path-like object.

class lzma.LZMAFile(filename=None, mode="r", *, format=None, check=-1, preset=None, filters=None)

Open an LZMA-compressed file in binary mode.

An LZMAFile can wrap an already-open file object, or operate directly on a named file. The filename argument specifies either the file object to wrap, or the name of the file to open (as a str, bytes or path-like object). When wrapping an existing file object, the wrapped file will not be closed when the LZMAFile is closed.

The mode argument can be either "r" for reading (default), "w" for overwriting, "x" for exclusive creation, or "a" for appending. These can equivalently be given as "rb", "wb", "xb" and "ab" respectively.

If filename is a file object (rather than an actual file name), a mode of "w" does not truncate the file, and is instead equivalent to "a".

When opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the format and filters arguments have the same meanings as for LZMADecompressor. In this case, the check and preset arguments should not be used.

When opening a file for writing, the format, check, preset and filters arguments have the same meanings as for LZMACompressor.
**LZMAFile** supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method is also provided:

**`peek(size=-1)`**

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the `size` argument is ignored).

**Note:** While calling `peek()` does not change the file position of the **LZMAFile**, it may change the position of the underlying file object (e.g. if the **LZMAFile** was constructed by passing a file object for `filename`).

*Changed in version 3.4:* Added support for the "x" and "xb" modes.

*Changed in version 3.5:* The `read()` method now accepts an argument of `None`.

*Changed in version 3.6:* Accepts a **path-like object**.
Compressing and decompressing data in memory

```python
class lzma.LZMACompressor(format=FORMAT_XZ, check=-1, preset=None, filters=None)
```

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see `compress()`.

The `format` argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ**: The `.xz` container format.
  
  This is the default format.

- **FORMAT_ALONE**: The legacy `.lzma` container format.
  
  This format is more limited than `.xz` – it does not support integrity checks or multiple filters.

- **FORMAT_RAW**: A raw data stream, not using any container format.
  
  This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using `FORMAT_AUTO` (see `LZMADecompressor`).

The `check` argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:
- **CHECK_NONE**: No integrity check. This is the default (and the only acceptable value) for FORMAT_ALONE and FORMAT_RAW.
- **CHECK_CRC32**: 32-bit Cyclic Redundancy Check.
- **CHECK_CRC64**: 64-bit Cyclic Redundancy Check. This is the default for FORMAT_XZ.
- **CHECK_SHA256**: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an **LZMAError** is raised.

The compression settings can be specified either as a preset compression level (with the `preset` argument), or in detail as a custom filter chain (with the `filters` argument).

The `preset` argument (if provided) should be an integer between 0 and 9 (inclusive), optionally OR-ed with the constant **PRESET_EXTREME**. If neither `preset` nor `filters` are given, the default behavior is to use PRESET_DEFAULT (preset level 6). Higher presets produce smaller output, but make the compression process slower.

**Note**: In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset 9 for example, the overhead for an **LZMACompressor** object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The `filters` argument (if provided) should be a filter chain specifier. See [Specifying custom filter chains](#) for details.

**compress(data)**

Compress `data` (a `bytes` object), returning a `bytes` object containing compressed data for at least part of the input. Some
of data may be buffered internally, for use in later calls to compress() and flush(). The returned data should be concatenated with the output of any previous calls to compress().

**flush()**

Finish the compression process, returning a bytes object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.LZMADecompressor(format=FORMAT_AUTO, memlimit=None, filters=None)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see decompress().

The format argument specifies the container format that should be used. The default is FORMAT_AUTO, which can decompress both .xz and .lzma files. Other possible values are FORMAT_XZ, FORMAT_ALONE, and FORMAT_RAW.

The memlimit argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an LZMAError if it is not possible to decompress the input within the given memory limit.

The filters argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if format is FORMAT_RAW, but should not be used for other formats.
See See [Specifying custom filter chains](#) for more information about filter chains.

**Note:** This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `LZMAFile`. To decompress a multi-stream input with `LZMADecompressor`, you must create a new decompressor for each stream.

### `decompress(data, max_length=-1)`

Decompress `data` (a bytes-like object), returning uncompressed data as bytes. Some of `data` may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If `max_length` is nonnegative, returns at most `max_length` bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide `data` as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than `max_length` bytes, or because `max_length` was negative), the `needs_input` attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an `EOFError`. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

*Changed in version 3.5:* Added the `max_length` parameter.
check
The ID of the integrity check used by the input stream. This may be CHECK_UNKNOWN until enough of the input has been decoded to determine what integrity check it uses.

eof
True if the end-of-stream marker has been reached.

unused_data
Data found after the end of the compressed stream.

   Before the end of the stream is reached, this will be b"".

needs_input
False if the decompress() method can provide more decompressed data before requiring new uncompressed input.

   New in version 3.5.

lzma.compress(data, format=FORMAT_XZ, check=-1, preset=None, filters=None)
   Compress data (a bytes object), returning the compressed data as a bytes object.

   See LZMACompressor above for a description of the format, check, preset and filters arguments.

lzma.decompress(data, format=FORMAT_AUTO, memlimit=None, filters=None)
   Decompress data (a bytes object), returning the uncompressed data as a bytes object.

   If data is the concatenation of multiple distinct compressed
streams, decompress all of these streams, and return the concatenation of the results.

See LZMADecompressor above for a description of the format, memlimit and filters arguments.
Miscellaneous

```python
lzma.is_check_supported(check)
```
Returns true if the given integrity check is supported on this system.

CHECK_NONE and CHECK_CRC32 are always supported. CHECK_CRC64 and CHECK_SHA256 may be unavailable if you are using a version of liblzma that was compiled with a limited feature set.
Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key "id", and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- Compression filters:
  - FILTER_LZMA1 (for use with FORMAT_ALONE)
  - FILTER_LZMA2 (for use with FORMAT_XZ and FORMAT_RAW)
- Delta filter:
  - FILTER_DELTA
- Branch-Call-Jump (BCJ) filters:
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- preset: A compression preset to use as a source of default values for options that are not specified explicitly.
- dict_size: Dictionary size in bytes. This should be
between 4 KiB and 1.5 GiB (inclusive).

- \(lc\): Number of literal context bits.
- \(lp\): Number of literal position bits. The sum \(lc + lp\) must be at most 4.
- \(pb\): Number of position bits; must be at most 4.
- \(mode\): \texttt{MODE\_FAST} or \texttt{MODE\_NORMAL}.
- \(nice\_len\): What should be considered a “nice length” for a match. This should be 273 or less.
- \(mf\): What match finder to use – \texttt{MF\_HC3}, \texttt{MF\_HC4}, \texttt{MF\_BT2}, \texttt{MF\_BT3}, or \texttt{MF\_BT4}.
- \(depth\): Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, \(dist\). This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, \(start\_offset\). This specifies the address that should be mapped to the beginning of the input data. The default is 0.
Examples

Reading in a compressed file:

```python
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```python
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```python
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```python
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:
import lzma

with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")

with lzma.open(f, "w") as lzf:
    lzf.write(b"This *will* be compressed\n")

f.write(b"Not compressed\n")

Creating a compressed file using a custom filter chain:

import lzma

my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME}
]

with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

**exception zipfile.BadZipFile**

The error raised for bad ZIP files.

*New in version 3.2.*

**exception zipfile.BadZipfile**

Alias of BadZipFile, for compatibility with older Python versions.

*Deprecated since version 3.2.*

**exception zipfile.LargeZipFile**

The error raised when a ZIP file would require ZIP64 functionality
but that has not been enabled.

```
class zipfile.ZipFile
    The class for reading and writing ZIP files. See section ZipFile Objects for constructor details.

class zipfile.PyZipFile
    Class for creating ZIP archives containing Python libraries.

class zipfile.ZipInfo(filename='NoName', date_time=(1980, 1, 1, 0, 0, 0))
    Class used to represent information about a member of an archive. Instances of this class are returned by the getinfo() and infolist() methods of ZipFile objects. Most users of the zipfile module will not need to create these, but only use those created by this module. filename should be the full name of the archive member, and date_time should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section ZipInfo Objects.

zipfile.is_zipfile(filename)
    Returns True if filename is a valid ZIP file based on its magic number, otherwise returns False. filename may be a file or file-like object too.

    Changed in version 3.1: Support for file and file-like objects.

zipfile.ZIP_STORED
    The numeric constant for an uncompressed archive member.

zipfile.ZIP_DEFLATED
    The numeric constant for the usual ZIP compression method. This requires the zlib module.
zipfile. **ZIP_BZIP2**

The numeric constant for the BZIP2 compression method. This requires the `bz2` module.

*New in version 3.3.*

zipfile. **ZIP_LZMA**

The numeric constant for the LZMA compression method. This requires the `lzma` module.

*New in version 3.3.*

**Note:** The ZIP file format specification has included support for bzip2 compression since 2001, and for LZMA compression since 2006. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

**See also:**

**PKZIP Application Note**

Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

**Info-ZIP Home Page**

Information about the Info-ZIP project’s ZIP archive programs and development libraries.
ZipFile Objects

```python
class zipfile.ZipFile(file, mode='r', compression=ZIP_STORED, allowZip64=True, compresslevel=None)
```

Open a ZIP file, where `file` can be a path to a file (a string), a file-like object or a path-like object.

The `mode` parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If `mode` is `'x'` and `file` refers to an existing file, a `FileExistsError` will be raised. If `mode` is `'a'` and `file` refers to an existing ZIP file, then additional files are added to it. If `file` does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If `mode` is `'a'` and the file does not exist at all, it is created. If `mode` is `'r'` or `'a'`, the file should be seekable.

`compression` is the ZIP compression method to use when writing the archive, and should be `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA`; unrecognized values will cause `NotImplementedError` to be raised. If `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA` is specified but the corresponding module (`zlib`, `bz2` or `lzma`) is not available, `RuntimeError` is raised. The default is `ZIP_STORED`.

If `allowZip64` is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` zipfile will raise an exception when the ZIP file would require ZIP64 extensions.
The `compresslevel` parameter controls the compression level to use when writing files to the archive. When using `ZIP_STORED` or `ZIP_LZMA` it has no effect. When using `ZIP_DEFLATED` integers 0 through 9 are accepted (see `zlib` for more information). When using `ZIP_BZIP2` integers 1 through 9 are accepted (see `bz2` for more information).

If the file is created with mode `'w'`, `'x'` or `'a'` and then closed without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, `myzip` is closed after the `with` statement’s suite is finished—even if an exception occurs:

```python
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

*New in version 3.2:* Added the ability to use `ZipFile` as a context manager.

*Changed in version 3.3:* Added support for `bz2` and `lzma` compression.

*Changed in version 3.4:* ZIP64 extensions are enabled by default.

*Changed in version 3.5:* Added support for writing to unseekable streams. Added support for the `'x'` mode.

*Changed in version 3.6:* Previously, a plain `RuntimeError` was raised for unrecognized compression values.

*Changed in version 3.6.2:* The `file` parameter accepts a path-like
object.

*Changed in version 3.7:* Add the *compresslevel* parameter.

**ZipFile**.*close()*

Close the archive file. You must call *close()* before exiting your program or essential records will not be written.

**ZipFile**.*getinfo(name)*

Return a *ZipInfo* object with information about the archive member *name*. Calling *getinfo()* for a name not currently contained in the archive will raise a *KeyError*.

**ZipFile**.*infolist()*

Return a list containing a *ZipInfo* object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

**ZipFile**.*namelist()*

Return a list of archive members by name.

**ZipFile**.*open(name, mode='r', pwd=None, *, force_zip64=False)*

Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a *ZipInfo* object. The *mode* parameter, if included, must be *'r'* (the default) or *'w'*.*pwd* is the password used to decrypt encrypted ZIP files.

*open()* is also a context manager and therefore supports the *with* statement:

```python
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```
With `mode 'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: `read()`, `readline()`, `iterlines()`, `seek()`, `tell()`, `__iter__()`, `__next__()`. These objects can operate independently of the ZipFile.

With `mode='w'`, a writable file handle is returned, which supports the `write()` method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a `ValueError`.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass `force_zip64=True` to ensure that the header format is capable of supporting large files. If the file size is known in advance, construct a `ZipInfo` object with `file_size` set, and use that as the `name` parameter.

**Note:** The `open()`, `read()` and `extract()` methods can take a filename or a `ZipInfo` object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

*Changed in version 3.6:* Removed support of `mode='U'`. Use `io.TextIOWrapper` for reading compressed text files in `universal newlines` mode.

*Changed in version 3.6:* `open()` can now be used to write files into the archive with the `mode='w'` option.

*Changed in version 3.6:* Calling `open()` on a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

`ZipFile.extract(member, path=None, pwd=None)`
Extract a member from the archive to the current working directory; `member` must be its full name or a `ZipInfo` object. Its file information is extracted as accurately as possible. `path` specifies a different directory to extract to. `member` can be a filename or a `ZipInfo` object. `pwd` is the password used for encrypted files.

Returns the normalized path created (a directory or new file).

**Note:** If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on Windows. And all ".." components in a member filename will be removed, e.g.: `../../foo../../ba..r` becomes `foo../ba..r`. On Windows illegal characters (`:`, `<`, `>`, `|`, `", ``, `?`, and `*`) replaced by underscore (`_`).

*Changed in version 3.6:* Calling `extract()` on a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

*Changed in version 3.6.2:* The `path` parameter accepts a path-like object.

`ZipFile.extractall(path=None, members=None, pwd=None)`

Extract all members from the archive to the current working directory. `path` specifies a different directory to extract to. `members` is optional and must be a subset of the list returned by `namelist()`. `pwd` is the password used for encrypted files.

**Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of
path, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..". This module attempts to prevent that. See extract() note.

*Changed in version 3.6:* Calling extractall() on a closed ZipFile will raise a ValueError. Previously, a RuntimeError was raised.

*Changed in version 3.6.2:* The path parameter accepts a path-like object.

ZipFile.printdir()

Print a table of contents for the archive to sys.stdout.

ZipFile.setpassword(pwd)

Set pwd as default password to extract encrypted files.

ZipFile.read(name, pwd=None)

Return the bytes of the file name in the archive. name is the name of the file in the archive, or a ZipInfo object. The archive must be open for read or append. pwd is the password used for encrypted files and, if specified, it will override the default password set with setpassword(). Calling read() on a ZipFile that uses a compression method other than ZIP_STORED, ZIP_DEFLATED, ZIP_BZIP2 or ZIP_LZMA will raise a NotImplementedError. An error will also be raised if the corresponding compression module is not available.

*Changed in version 3.6:* Calling read() on a closed ZipFile will raise a ValueError. Previously, a RuntimeError was raised.

ZipFile.testzip()

Read all the files in the archive and check their CRC's and file
headers. Return the name of the first bad file, or else return None.

Changed in version 3.6: Calling `testzip()` on a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

`ZipFile.write(filename, arcname=None, compress_type=None, compresslevel=None)`

Write the file named `filename` to the archive, giving it the archive name `arcname` (by default, this will be the same as `filename`, but without a drive letter and with leading path separators removed). If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry. Similarly, `compresslevel` will override the constructor if given. The archive must be open with mode `'w'`, `'x'` or `'a'`.

**Note:** Archive names should be relative to the archive root, that is, they should not start with a path separator.

**Note:** If `arcname` (or `filename`, if `arcname` is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

Changed in version 3.6: Calling `write()` on a ZipFile created with mode `'r'` or a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

`ZipFile.writestr(zinfo_or_arcname, data, compress_type=None, compresslevel=None)`

Write a file into the archive. The contents is `data`, which may be either a `str` or a `bytes` instance; if it is a `str`, it is encoded as UTF-8 first. `zinfo_or_arcname` is either the file name it will be given
in the archive, or a `ZipInfo` instance. If it's an instance, at least
the filename, date, and time must be given. If it's a name, the date
and time is set to the current date and time. The archive must be
opened with mode `w`, `x` or `a`.

If given, `compress_type` overrides the value given for the
`compression` parameter to the constructor for the new entry, or in
the `zinfo_or_arcname` (if that is a `ZipInfo` instance). Similarly,
`compresslevel` will override the constructor if given.

**Note:** When passing a `ZipInfo` instance as the
`zinfo_or_arcname` parameter, the compression method used will
be that specified in the `compress_type` member of the given
`ZipInfo` instance. By default, the `ZipInfo` constructor sets this
member to `ZIP_STORED`.

*Changed in version 3.2:* The `compress_type` argument.

*Changed in version 3.6:* Calling `writestr()` on a ZipFile created
with mode `r` or a closed ZipFile will raise a `ValueError`. Previously,
a `RuntimeError` was raised.

The following data attributes are also available:

**ZipFile.** `filename`
Name of the ZIP file.

**ZipFile.** `debug`
The level of debug output to use. This may be set from 0 (the
default, no output) to 3 (the most output). Debugging information is
written to `sys.stdout`.

**ZipFile.** `comment`
The comment associated with the ZIP file as a bytes object. If assigning a comment to a ZipFile instance created with mode 'w', 'x' or 'a', it should be no longer than 65535 bytes. Comments longer than this will be truncated.
PyZipFile Objects

The **PyZipFile** constructor takes the same parameters as the **ZipFile** constructor, and one additional parameter, *optimize*.

```python
class zipfile.PyZipFile(file, mode='r',
    compression=ZIP_STORED, allowZip64=True, optimize=-1)
```

*New in version 3.2:* The *optimize* parameter.

*Changed in version 3.4:* ZIP64 extensions are enabled by default.

Instances have one method in addition to those of **ZipFile** objects:

```python
writepy(pathname, basename='', filterfunc=None)
```

Search for files *.py* and add the corresponding file to the archive.

If the *optimize* parameter to **PyZipFile** was not given or -1, the corresponding file is a *.pyc* file, compiling if necessary.

If the *optimize* parameter to **PyZipFile** was 0, 1 or 2, only files with that optimization level (see compile()) are added to the archive, compiling if necessary.

If *pathname* is a file, the filename must end with *.py*, and just the (corresponding *.pyc*) file is added at the top level (no path information). If *pathname* is a file that does not end with *.py*, a **RuntimeError** will be raised. If it is a directory, and the directory is not a package directory, then all the files *.pyc* are added at the top level. If the directory is a package directory, then all *.pyc* are added under the package name as a file.
path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

*basename* is intended for internal use only.

*filterfunc*, if given, must be a function taking a single string argument. It will be passed each path (including each individual full file path) before it is added to the archive. If *filterfunc* returns a false value, the path will not be added, and if it is a directory its contents will be ignored. For example, if our test files are all either in *test* directories or start with the string *test_*, we can use a *filterfunc* to exclude them:

```python
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

The *writepy()* method makes archives with file names like this:

```
string.pyc            # Top level name
|test/___init____.pyc # Package directory
|test/testall.pyc     # Module test.test
|test/bogus/___init____.pyc # Subpackage directory
|test/bogus/myfile.pyc # Submodule test.bogus
```

*New in version 3.4*: The *filterfunc* parameter.

*Changed in version 3.6.2*: The *pathname* parameter accepts a path-like object.
Changed in version 3.7: Recursion sorts directory entries.
ZipInfo Objects

Instances of the `ZipInfo` class are returned by the `getinfo()` and `infolist()` methods of `ZipFile` objects. Each object stores information about a single member of the ZIP archive.

There is one classmethod to make a `ZipInfo` instance for a filesystem file:

```
classmethod ZipInfo.from_file(filename, arcname=None)
```

Construct a `ZipInfo` instance for a file on the filesystem, in preparation for adding it to a zip file.

- `filename` should be the path to a file or directory on the filesystem.
- If `arcname` is specified, it is used as the name within the archive. If `arcname` is not specified, the name will be the same as `filename`, but with any drive letter and leading path separators removed.

*New in version 3.6.*

*Changed in version 3.6.2:* The `filename` parameter accepts a path-like object.

Instances have the following methods and attributes:

```
ZipInfo.is_dir()
```

Return `True` if this archive member is a directory.

This uses the entry's name: directories should always end with `/`.

*New in version 3.6.*
ZipInfo. **filename**
Name of the file in the archive.

**ZipInfo. date_time**
The time and date of the last modification to the archive member. This is a tuple of six values:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Year (&gt;= 1980)</td>
</tr>
<tr>
<td>1</td>
<td>Month (one-based)</td>
</tr>
<tr>
<td>2</td>
<td>Day of month (one-based)</td>
</tr>
<tr>
<td>3</td>
<td>Hours (zero-based)</td>
</tr>
<tr>
<td>4</td>
<td>Minutes (zero-based)</td>
</tr>
<tr>
<td>5</td>
<td>Seconds (zero-based)</td>
</tr>
</tbody>
</table>

**Note:** The ZIP file format does not support timestamps before 1980.

**ZipInfo. compress_type**
Type of compression for the archive member.

**ZipInfo. comment**
Comment for the individual archive member as a `bytes` object.

**ZipInfo. extra**
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this `bytes` object.

**ZipInfo. create_system**
System which created ZIP archive.
ZipInfo.create_version
PKZIP version which created ZIP archive.

ZipInfo.extract_version
PKZIP version needed to extract archive.

ZipInfo.reserved
Must be zero.

ZipInfo.flag_bits
ZIP flag bits.

ZipInfo.volume
Volume number of file header.

ZipInfo.internal_attr
Internal attributes.

ZipInfo.external_attr
External file attributes.

ZipInfo.header_offset
Byte offset to the file header.

ZipInfo.CRC
CRC-32 of the uncompressed file.

ZipInfo.compress_size
Size of the compressed data.

ZipInfo.file_size
Size of the uncompressed file.
Command-Line Interface

The `zipfile` module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the `-e` option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the `-l` option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

- `-l < zipfile>`
- `--list < zipfile>`
  List files in a zipfile.

- `-c < zipfile> < source1 > ... < sourceN >`
- `--create < zipfile> < source1 > ... < sourceN >`
Create zipfile from source files.

- e  <zipfile>  <output_dir>
--extract  <zipfile>  <output_dir>
   Extract zipfile into target directory.

- t  <zipfile>
--test  <zipfile>
   Test whether the zipfile is valid or not.
The `tarfile` module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the `zipfile` module to read or write `.zip` files, or the higher-level functions in `shutil`.

Some facts and figures:

- reads and writes `gzip`, `bz2` and `lzma` compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including `longname` and `longlink` extensions, read-only support for all variants of the `sparse` extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

*Changed in version 3.3:* Added support for `lzma` compression.

`tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)`

Return a `TarFile` object for the pathname `name`. For detailed information on `TarFile` objects and the keyword arguments that
are allowed, see **TarFile Objects**.

`mode` has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

<table>
<thead>
<tr>
<th>mode</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'r'</code> or <code>'r:*'</code></td>
<td>Open for reading with transparent compression (recommended).</td>
</tr>
<tr>
<td><code>'r:'</code></td>
<td>Open for reading exclusively without compression.</td>
</tr>
<tr>
<td><code>'r:gz'</code></td>
<td>Open for reading with gzip compression.</td>
</tr>
<tr>
<td><code>'r:bz2'</code></td>
<td>Open for reading with bzip2 compression.</td>
</tr>
<tr>
<td><code>'r:xz'</code></td>
<td>Open for reading with lzma compression.</td>
</tr>
<tr>
<td><code>'x'</code> or <code>'x:'</code></td>
<td>Create a tarfile exclusively without compression. Raise an <code>FileExistsError</code> exception if it already exists.</td>
</tr>
<tr>
<td><code>'x:gz'</code></td>
<td>Create a tarfile with gzip compression. Raise an <code>FileExistsError</code> exception if it already exists.</td>
</tr>
<tr>
<td><code>'x:bz2'</code></td>
<td>Create a tarfile with bzip2 compression. Raise an <code>FileExistsError</code> exception if it already exists.</td>
</tr>
<tr>
<td><code>'x:xz'</code></td>
<td>Create a tarfile with lzma compression. Raise an <code>FileExistsError</code> exception if it already exists.</td>
</tr>
<tr>
<td><code>'a'</code> or <code>'a:'</code></td>
<td>Open for appending with no compression. The file is created if it does not exist.</td>
</tr>
<tr>
<td><code>'w'</code> or <code>'w:'</code></td>
<td>Open for uncompressed writing.</td>
</tr>
<tr>
<td><code>'w:gz'</code></td>
<td>Open for gzip compressed writing.</td>
</tr>
</tbody>
</table>
Open for bzip2 compressed writing.

Open for lzma compressed writing.

Note that 'a:gz', 'a:bz2' or 'a:xz' is not possible. If mode is not suitable to open a certain (compressed) file for reading, ReadError is raised. Use mode 'r' to avoid this. If a compression method is not supported, CompressionError is raised.

If fileobj is specified, it is used as an alternative to a file object opened in binary mode for name. It is supposed to be at position 0.

For modes 'w:gz', 'r:gz', 'w:bz2', 'r:bz2', 'x:gz', 'x:bz2', tarfile.open() accepts the keyword argument compresslevel (default 9) to specify the compression level of the file.

For special purposes, there is a second format for mode: 'filemode[[compression]]'. tarfile.open() will return a TarFile object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, fileobj may be any object that has a read() or write() method (depending on the mode). bufsize specifies the blocksize and defaults to $20 * 512$ bytes. Use this variant in combination with e.g. sys.stdin, a socket file object or a tape device. However, such a TarFile object is limited in that it does not allow random access, see Examples. The currently possible modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r</td>
<td>*'</td>
</tr>
<tr>
<td>Mode</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>'r'</td>
<td>Open a stream of uncompressed tar blocks for reading.</td>
</tr>
<tr>
<td>'r</td>
<td>gz'</td>
</tr>
<tr>
<td>'r</td>
<td>bz2'</td>
</tr>
<tr>
<td>'r</td>
<td>xz'</td>
</tr>
<tr>
<td>'w'</td>
<td>Open an uncompressed stream for writing.</td>
</tr>
<tr>
<td>'w</td>
<td>gz'</td>
</tr>
<tr>
<td>'w</td>
<td>bz2'</td>
</tr>
<tr>
<td>'w</td>
<td>xz'</td>
</tr>
</tbody>
</table>

*Changed in version 3.5:* The 'x' (exclusive creation) mode was added.

*Changed in version 3.6:* The `name` parameter accepts a path-like object.

**class tarfile.TarFile**

Class for reading and writing tar archives. Do not use this class directly: use `tarfile.open()` instead. See TarFile Objects.

**tarfile.is_tarfile(name)**

Return `True` if `name` is a tar archive file, that the `tarfile` module can read.

The `tarfile` module defines the following exceptions:

**exception tarfile.TarError**

Base class for all `tarfile` exceptions.

**exception tarfile.ReadError**

Is raised when a tar archive is opened, that either cannot be
handled by the `tarfile` module or is somehow invalid.

`exception` `tarfile.CompressionError`

Is raised when a compression method is not supported or when the data cannot be decoded properly.

`exception` `tarfile.StreamError`

Is raised for the limitations that are typical for stream-like `TarFile` objects.

`exception` `tarfile.ExtractError`

Is raised for `non-fatal` errors when using `TarFile.extract()`, but only if `TarFile.errorlevel==2`.

`exception` `tarfile.HeaderError`

Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

The following constants are available at the module level:

`tarfile.ENCODING`

The default character encoding: 'utf-8' on Windows, the value returned by `sys.getfilesystemencoding()` otherwise.

Each of the following constants defines a tar archive format that the `tarfile` module is able to create. See section `Supported tar formats` for details.

`tarfile.USTAR_FORMAT`

POSIX.1-1988 (ustar) format.

`tarfile.GNU_FORMAT`

GNU tar format.
tarfile. **PAX_FORMAT**

POSIX.1-2001 (pax) format.

**tarfile. DEFAULT_FORMAT**

The default format for creating archives. This is currently **GNU_FORMAT**.

---

**See also:**

**Module zipfile**

Documentation of the **zipfile** standard module.

**Archiving operations**

Documentation of the higher-level archiving facilities provided by the standard **shutil** module.

**GNU tar manual, Basic Tar Format**

Documentation for tar archive files, including GNU tar extensions.
**TarFile Objects**

The **TarFile** object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a **TarInfo** object, see **TarInfo Objects** for details.

A **TarFile** object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the **Examples** section for a use case.

*New in version 3.2:* Added support for the context management protocol.

```python
class tarfile.TarFile(name=None, mode='r', fileobj=None,
                      format=DEFAULT_FORMAT, tarinfo=TarInfo, dereference=False,
                      ignore_zeros=False, encoding=ENCODING, errors='surrogateescape',
                      pax_headers=None, debug=0, errorlevel=0)
```

All following arguments are optional and can be accessed as instance attributes as well.

- **name** is the pathname of the archive. **name** may be a path-like object. It can be omitted if **fileobj** is given. In this case, the file object’s name attribute is used if it exists.

- **mode** is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.
If `fileobj` is given, it is used for reading or writing data. If it can be determined, `mode` is overridden by `fileobj`'s mode. `fileobj` will be used from position 0.

**Note:** `fileobj` is not closed, when `TarFile` is closed.

`format` controls the archive format. It must be one of the constants `USTAR_FORMAT`, `GNU_FORMAT` or `PAX_FORMAT` that are defined at module level.

The `tarinfo` argument can be used to replace the default `TarInfo` class with a different one.

If `dereference` is `False`, add symbolic and hard links to the archive. If it is `True`, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If `ignore_zeros` is `False`, treat an empty block as the end of the archive. If it is `True`, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

`debug` can be set from 0 (no debug messages) up to 3 (all debug messages). The messages are written to `sys.stderr`.

If `errorlevel` is 0, all errors are ignored when using `TarFile.extract()`. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all `fatal` errors are raised as `OSError` exceptions. If 2, all `non-fatal` errors are raised as `TarError` exceptions as well.

The `encoding` and `errors` arguments define the character encoding
to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section Unicode issues for in-depth information.

The `pax_headers` argument is an optional dictionary of strings which will be added as a pax global header if `format` is `PAX_FORMAT`.

*Changed in version 3.2:* Use `surrogateescape` as the default for the `errors` argument.

*Changed in version 3.5:* The `x` (exclusive creation) mode was added.

*Changed in version 3.6:* The `name` parameter accepts a path-like object.

**classmethod** `TarFile.open(...)`

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

`TarFile.getmember(name)`

Return a `TarInfo` object for member `name`. If `name` can not be found in the archive, `KeyError` is raised.

**Note:** If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

`TarFile.getmembers()`

Return the members of the archive as a list of `TarInfo` objects. The list has the same order as the members in the archive.

`TarFile.getnames()`
Return the members as a list of their names. It has the same order as the list returned by `getmembers()`.

`TarFile.list(verbosetrue=True, *, members=None)`

Print a table of contents to `sys.stdout`. If `verbose` is `False`, only the names of the members are printed. If it is `True`, output similar to that of `ls -l` is produced. If optional `members` is given, it must be a subset of the list returned by `getmembers()`.

*Changed in version 3.5: Added the `members` parameter.*

`TarFile.next()`

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return `None` if there is no more available.

`TarFile.extractall(path=".", members=None, *, numeric_owner=False)`

Extract all members from the archive to the current working directory or directory `path`. If optional `members` is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If `numeric_owner` is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

**Warning:** Never extract archives from untrusted sources without
prior inspection. It is possible that files are created outside of path, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

*Changed in version 3.5:* Added the `numeric_owner` parameter.

*Changed in version 3.6:* The `path` parameter accepts a path-like object.

```python
TarFile.extract(member, path="", set_attrs=True, *, numeric_owner=False)
```

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. `member` may be a filename or a `TarInfo` object. You can specify a different directory using `path`. `path` may be a path-like object. File attributes (owner, mtime, mode) are set unless `set_attrs` is false.

If `numeric_owner` is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

**Note:** The `extract()` method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.

**Warning:** See the warning for `extractall()`.

*Changed in version 3.2:* Added the `set_attrs` parameter.

*Changed in version 3.5:* Added the `numeric_owner` parameter.

*Changed in version 3.6:* The `path` parameter accepts a path-like
object.

TarFile.extractfile(member)

Extract a member from the archive as a file object. member may be a filename or a TarInfo object. If member is a regular file or a link, an io.BufferedReader object is returned. Otherwise, None is returned.

*Changed in version 3.3:* Return an io.BufferedReader object.

TarFile.add(name, arcname=None, recursive=True, *, filter=None)

Add the file name to the archive. name may be any type of file (directory, fifo, symbolic link, etc.). If given, arcname specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting recursive to False. Recursion adds entries in sorted order. If filter is given, it should be a function that takes a TarInfo object argument and returns the changed TarInfo object. If it instead returns None the TarInfo object will be excluded from the archive. See Examples for an example.

*Changed in version 3.2:* Added the filter parameter.

*Changed in version 3.7:* Recursion adds entries in sorted order.

TarFile.addfile(tarinfo, fileobj=None)

Add the TarInfo object tarinfo to the archive. If fileobj is given, it should be a binary file, and tarinfo.size bytes are read from it and added to the archive. You can create TarInfo objects directly, or by using gettarinfo().

TarFile.gettarinfo(name=None, arcname=None, fileobj=None)
Create a `TarInfo` object from the result of `os.stat()` or equivalent on an existing file. The file is either named by `name`, or specified as a `file object fileobj` with a file descriptor. `name` may be a path-like object. If given, `arcname` specifies an alternative name for the file in the archive, otherwise, the name is taken from `fileobj`'s `name` attribute, or the `name` argument. The name should be a text string.

You can modify some of the `TarInfo`'s attributes before you add it using `addfile()`. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as `size` may need modifying. This is the case for objects such as `GzipFile`. The `name` may also be modified, in which case `arcname` could be a dummy string.

*Changed in version 3.6:* The `name` parameter accepts a path-like object.

**TarFile**. `close()`  
Close the `TarFile`. In write mode, two finishing zero blocks are appended to the archive.

**TarFile**. `pax_headers`  
A dictionary containing key-value pairs of pax global headers.
TarInfo Objects

A TarInfo object represents one member in a TarFile. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does not contain the file's data itself.

TarInfo objects are returned by TarFile's methods getmember(), getmembers() and gettarinfo().

```
class tarfile.TarInfo(name="")
    Create a TarInfo object.

classmethod TarInfo.frombuf(buf, encoding, errors)
    Create and return a TarInfo object from string buffer buf.

    Raises HeaderError if the buffer is invalid.

classmethod TarInfo.fromtarfile(tarfile)
    Read the next member from the TarFile object tarfile and return it as a TarInfo object.

TarInfo.tobuf(format=DEFAULT_FORMAT, encoding=ENCODING, errors='surrogateescape')
    Create a string buffer from a TarInfo object. For information on the arguments see the constructor of the TarFile class.

    Changed in version 3.2: Use 'surrogateescape' as the default for the errors argument.
```

A TarInfo object has the following public data attributes:
TarInfo. **name**
    Name of the archive member.

TarInfo. **size**
    Size in bytes.

TarInfo. **mtime**
    Time of last modification.

TarInfo. **mode**
    Permission bits.

TarInfo. **type**
    File type. *type* is usually one of these constants: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE. To determine the type of a TarInfo object more conveniently, use the is*() methods below.

TarInfo. **linkname**
    Name of the target file name, which is only present in TarInfo objects of type LNKTYPE and SYMTYPE.

TarInfo. **uid**
    User ID of the user who originally stored this member.

TarInfo. **gid**
    Group ID of the user who originally stored this member.

TarInfo. **uname**
    User name.

TarInfo. **gname**
Group name.

TarInfo.pax_headers
A dictionary containing key-value pairs of an associated pax extended header.

A TarInfo object also provides some convenient query methods:

TarInfo.isfile()
Return True if the Tarinfo object is a regular file.

TarInfo.isreg()
Same as isfile().

TarInfo.isdir()
Return True if it is a directory.

TarInfo.issym()
Return True if it is a symbolic link.

TarInfo.islnk()
Return True if it is a hard link.

TarInfo.ischr()
Return True if it is a character device.

TarInfo.isblk()
Return True if it is a block device.

TarInfo.isfifo()
Return True if it is a FIFO.

TarInfo.isdev()
Return True if it is one of character device, block device or FIFO.
Command-Line Interface

New in version 3.4.

The `tarfile` module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the `-e` option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory’s name:

```
$ python -m tarfile -e monty.tar other-dir/
```

For a list of the files in a tar archive, use the `-l` option:

```
$ python -m tarfile -l monty.tar
```

Command-line options
-l <tarfile>
--list <tarfile>
  List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
  Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
  Extract tarfile into the current directory if output_dir is not specified.

-t <tarfile>
--test <tarfile>
  Test whether the tarfile is valid or not.

-v, --verbose
  Verbose output.
Examples

How to extract an entire tar archive to the current working directory:

```python
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with `TarFile.extractall()` using a generator function instead of a list:

```python
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py"
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```python
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:
```python
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```python
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the `filter` parameter in `TarFile.add()`:

```python
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```
Supported tar formats

There are three tar formats that can be created with the `tarfile` module:

- The POSIX.1-1988 ustar format (`USTAR_FORMAT`). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.

- The GNU tar format (`GNU_FORMAT`). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. `tarfile` fully supports the GNU tar extensions for long names, sparse file support is read-only.

- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

The `pax` format is an extension to the existing `ustar` format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in `UTF-8` for portability reasons.

There are some more variants of the tar format which can be read, but not created:
• The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.

• The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.
Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a UTF-8 system cannot be read correctly on a Latin-1 system if it contains non-ASCII characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding UTF-8.

The details of character conversion in tarfile are controlled by the encoding and errors keyword arguments of the TarFile class.

encoding defines the character encoding to use for the metadata in the archive. The default value is sys.getfilesystemencoding() or 'ascii' as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If encoding is not set appropriately, this conversion may fail.

The errors argument defines how characters are treated that cannot be converted. Possible values are listed in section Error Handlers. The default scheme is 'surrogateescape' which Python also uses for its file system calls, see File Names, Command Line Arguments, and Environment Variables.

In case of PAX_FORMAT archives, encoding is generally not needed
because all the metadata is stored using UTF-8. encoding is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.
File Formats

The modules described in this chapter parse various miscellaneous file formats that aren’t markup languages and are not related to e-mail.

- **csv** — CSV File Reading and Writing
  - Module Contents
  - Dialects and Formatting Parameters
  - Reader Objects
  - Writer Objects
  - Examples

- **configparser** — Configuration file parser
  - Quick Start
  - Supported Datatypes
  - Fallback Values
  - Supported INI File Structure
  - Interpolation of values
  - Mapping Protocol Access
  - Customizing Parser Behaviour
  - Legacy API Examples
  - ConfigParser Objects
  - RawConfigParser Objects
  - Exceptions

- **netrc** — netrc file processing
  - netrc Objects

- **xdrlib** — Encode and decode XDR data
  - Packer Objects
  - Unpacker Objects
  - Exceptions

- **plistlib** — Generate and parse Mac OS X .plist files
  - Examples
**csv — CSV File Reading and Writing**

**Source code:** Lib/csv.py

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in RFC 4180. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module's **reader** and **writer** objects read and write sequences. Programmers can also read and write data in dictionary form using the **DictReader** and **DictWriter** classes.
See also:

**PEP 305 - CSV File API**

The Python Enhancement Proposal which proposed this addition to Python.
Module Contents

The csv module defines the following functions:

```python
csv.reader(csvfile, dialect='excel', **fmtparams)
```

Return a reader object which will iterate over lines in the given `csvfile`. `csvfile` can be any object which supports the iterator protocol and returns a string each time its `__next__()` method is called — file objects and list objects are both suitable. If `csvfile` is a file object, it should be opened with `newline=''`. [1] An optional `dialect` parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the `QUOTE_NONNUMERIC` format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

```python
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=','
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
```
Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. `csvfile` can be any object with a `write()` method. If `csvfile` is a file object, it should be opened with `newline=''` [1]. An optional `dialect` parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters. To make it as easy as possible to interface with modules which implement the DB API, the value `None` is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetchall()` call. All other non-string data are stringified with `str()` before being written.

A short usage example:

```python
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                             quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```
`csv.register_dialect(name[, dialect[, **fmtparams]])`

Associate `dialect` with `name`. `name` must be a string. The dialect can be specified either by passing a sub-class of `Dialect`, or by `fmtparams` keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

`csv.unregister_dialect(name)`

Delete the dialect associated with `name` from the dialect registry. An `Error` is raised if `name` is not a registered dialect name.

`csv.get_dialect(name)`

Return the dialect associated with `name`. An `Error` is raised if `name` is not a registered dialect name. This function returns an immutable `Dialect`.

`csv.list_dialects()`

Return the names of all registered dialects.

`csv.field_size_limit([new_limit])`

Returns the current maximum field size allowed by the parser. If `new_limit` is given, this becomes the new limit.

The `csv` module defines the following classes:

```
csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
```

Create an object that operates like a regular reader but maps the information in each row to an `OrderedDict` whose keys are given by the optional `fieldnames` parameter.
The fieldnames parameter is a sequence. If fieldnames is omitted, the values in the first row of file f will be used as the fieldnames. Regardless of how the fieldnames are determined, the ordered dictionary preserves their original ordering.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by restkey (which defaults to None). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with None.

All other optional or keyword arguments are passed to the underlying reader instance.

Changed in version 3.6: Returned rows are now of type OrderedDict.

A short usage example:

```python
>>> import csv
>>> with open('names.csv', newline='') as csvfile
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
Eric Idle
John Cleese
```

```python
>>> print(row)
OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```
sequence of keys that identify the order in which values in the
dictionary passed to the writerow() method are written to file f.
The optional restval parameter specifies the value to be written if
the dictionary is missing a key in fieldnames. If the dictionary
passed to the writerow() method contains a key not found in
fieldnames, the optional extrasaction parameter indicates what
action to take. If it is set to 'raise', the default value, a
ValueError is raised. If it is set to 'ignore', extra values in the
dictionary are ignored. Any other optional or keyword arguments
are passed to the underlying writer instance.

Note that unlike the DictReader class, the fieldnames parameter
of the DictWriter class is not optional.

A short usage example:

```python
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Baker'}
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Sleepy'}
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Dreamer'})
```

class csv.Dialect

The Dialect class is a container class relied on primarily for its
attributes, which are used to define the parameters for a specific
reader or writer instance.

class csv.excel
The `excel` class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name 'excel'.

```python
class csv.excel_tab
```

The `excel_tab` class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name 'excel-tab'.

```python
class csv_unix_dialect
```

The `unix_dialect` class defines the usual properties of a CSV file generated on UNIX systems, i.e. using '\n' as line terminator and quoting all fields. It is registered with the dialect name 'unix'.

*New in version 3.2.*

```python
class csv.Sniffer
```

The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class provides two methods:

```python
sniff(sample, delimiters=None)
```

Analyze the given `sample` and return a `Dialect` subclass reflecting the parameters found. If the optional `delimiters` parameter is given, it is interpreted as a string containing possible valid delimiter characters.

```python
has_header(sample)
```

Analyze the sample text (presumed to be in CSV format) and return `True` if the first row appears to be a series of column headers.

An example for `Sniffer` use:
```python
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The `csv` module defines the following constants:

**csv.QUOTE_ALL**
Instructs `writer` objects to quote all fields.

**csv.QUOTE_MINIMAL**
Instructs `writer` objects to only quote those fields which contain special characters such as `delimiter`, `quotechar` or any of the characters in `lineterminator`.

**csv.QUOTE_NONNUMERIC**
Instructs `writer` objects to quote all non-numeric fields.

   Instructs the reader to convert all nonquoted fields to type `float`.

**csv.QUOTE_NONE**
Instructs `writer` objects to never quote fields. When the current `delimiter` occurs in output data it is preceded by the current `escapechar` character. If `escapechar` is not set, the writer will raise `Error` if any characters that require escaping are encountered.

   Instructs `reader` to perform no special processing of quote characters.

The `csv` module defines the following exception:

```python
exception csv.Error
```
Raised by any of the functions when an error is detected.
Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the `Dialect` class having a set of specific methods and a single `validate()` method. When creating `reader` or `writer` objects, the programmer can specify a string or a subclass of the `Dialect` class as the dialect parameter. In addition to, or instead of, the `dialect` parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the `Dialect` class.

Dialects support the following attributes:

**Dialect**. `delimiter`

A one-character string used to separate fields. It defaults to `',',`.

**Dialect**. `doublequote`

Controls how instances of `quotechar` appearing inside a field should themselves be quoted. When `True`, the character is doubled. When `False`, the `escapechar` is used as a prefix to the `quotechar`. It defaults to `True`.

On output, if `doublequote` is `False` and no `escapechar` is set, `Error` is raised if a `quotechar` is found in a field.

**Dialect**. `escapechar`

A one-character string used by the writer to escape the `delimiter` if quoting is set to `QUOTE_NONE` and the `quotechar` if `doublequote` is `False`. On reading, the `escapechar` removes any special meaning from the following character. It defaults to `None`, which disables
escaping.

**Dialect. lineterminator**

The string used to terminate lines produced by the *writer*. It defaults to '\r\n'.

**Note:** The *reader* is hard-coded to recognise either '\r' or '\n' as end-of-line, and ignores *lineterminator*. This behavior may change in the future.

**Dialect. quotechar**

A one-character string used to quote fields containing special characters, such as the *delimiter* or *quotechar*, or which contain new-line characters. It defaults to "".

**Dialect. quoting**

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

**Dialect. skipinitialspace**

When *True*, whitespace immediately following the *delimiter* is ignored. The default is *False*.

**Dialect. strict**

When *True*, raise exception *Error* on bad CSV input. The default is *False*. 
Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

csvreader.__next__()

    Return the next row of the reader’s iterable object as a list (if the object was returned from reader()) or a dict (if it is a DictReader instance), parsed according to the current dialect. Usually you should call this as next(reader).

Reader objects have the following public attributes:

csvreader.dialect

    A read-only description of the dialect in use by the parser.

csvreader.line_num

    The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

DictReader objects have the following public attribute:

csvreader.fieldnames

    If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.
Writer Objects

Writer objects (``DictWriter`` instances and objects returned by the ``writer()`` function) have the following public methods. A row must be an iterable of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through `str()` first) for DictWriter objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.``writerow``(row)

Write the row parameter to the writer’s file object, formatted according to the current dialect.

*Changed in version 3.5: Added support of arbitrary iterables.*

csvwriter.``writerows``(rows)

Write all elements in rows (an iterable of row objects as described above) to the writer's file object, formatted according to the current dialect.

Writer objects have the following public attribute:

csvwriter.``dialect``

A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

DictWriter.``writeheader``()

Write a row with the field names (as specified in the constructor).
New in version 3.2.
Examples

The simplest example of reading a CSV file:

```python
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Reading a file with an alternate format:

```python
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_ALL)
    for row in reader:
        print(row)
```

The corresponding simplest possible writing example is:

```python
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Since `open()` is used to open a CSV file for reading, the file will by default be decoded into unicode using the system default encoding (see `locale.getpreferredencoding()`). To decode a file using a different encoding, use the `encoding` argument of open:

```python
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
```
The same applies to writing in something other than the system default encoding: specify the encoding argument when opening the output file.

Registering a new dialect:

```python
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_ALL)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```python
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, e.lineno, e))
```

And while the module doesn’t directly support parsing strings, it can easily be done:

```python
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```
Footnotes

(1, 2) If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use \r\n linendings on write an extra \r will be added. It should always be safe to specify `newline=''`, since the csv module does its own (universal) newline handling.
This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

**Note:** This library does not interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

**See also:**

**Module** `shlex`
Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

**Module** `json`
The json module implements a subset of JavaScript syntax which can also be used for this purpose.
Quick Start

Let's take a very basic configuration file that looks like this:

```ini
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described in the following section. Essentially, the file consists of sections, each of which contains keys with values. `configparser` classes can read and write such files. Let's start by creating the above configuration file programmatically.

```python
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
... 'Compression': 'yes',
... 'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'  # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...     config.write(configfile)
```
As you can see, we can treat a config parser much like a dictionary. There are differences, outlined later, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```python
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...    print(key)
user
compressionlevel
serveraliveinterval
```
As we can see above, the API is pretty straightforward. The only bit of magic involves the DEFAULT section which provides default values for all other sections [1]. Note also that keys in sections are case-insensitive and stored in lowercase [1].
Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```python
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide `getboolean()`. This method is case-insensitive and recognizes Boolean values from 'yes'/'no', 'on'/'off', 'true'/'false' and '1'/'0' [1]. For example:

```python
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Apart from `getboolean()`, config parsers also provide equivalent `getint()` and `getfloat()` methods. You can register your own converters and customize the provided ones. [1]
Fallback Values

As with a dictionary, you can use a section’s get() method to provide fallback values:

```py
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `CompressionLevel` key was specified only in the `DEFAULT` section. If we try to get it from the section `topsecret.server.com`, we will always get the default, even if we specify a fallback:

```py
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level get() method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```py
>>> config.get('bitbucket.org', 'monster', ...
...   fallback='No such things as monsters'
'No such things as monsters'
```

The same `fallback` argument can be used with the `getint()`,
getfloat() and getboolean() methods, for example:

```python
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```
Supported INI File Structure

A configuration file consists of sections, each led by a [section] header, followed by key/value entries separated by a specific string (= or : by default [1]). By default, section names are case sensitive but keys are not [1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted, in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser’s mode, blank lines may be treated as parts of multiline values or ignored.

Configuration files may include comments, prefixed by specific characters (# and ; by default [1]). Comments may appear on their own on an otherwise empty line, possibly indented. [1]

For example:

**[Simple Values]**
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

**[All Values Are Strings]**
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: tr

**[Multiline Values]**
chorus: I'm a lumberjack, and I'm okay
        I sleep all night and I work all day

[No Values]
key_without_value
empty_string_value_here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

[Sections Can Be Indented]
can_values_be_as_well = True
does_that_mean_anything_special = False
purpose = formatting for readability
multiline_values = are
    handled just fine as
    long as they are indented
    deeper than the first line
    of a value
# Did I mention we can indent comments, too?
Interpolation of values

On top of the core functionality, ConfigParser supports interpolation. This means values can be preprocessed before returning them from get() calls.

```python
class configparser.BasicInterpolation
    The default implementation used by ConfigParser. It enables values to contain format strings which refer to other values in the same section, or values in the special default section [1]. Additional default values can be provided on initialization.
```

For example:

```ini
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures
```

In the example above, ConfigParser with interpolation set to BasicInterpolation() would resolve %(home_dir)s to the value of home_dir (/Users in this case). %(my_dir)s in effect would resolve to /Users/lumberjack. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With interpolation set to None, the parser would simply return %(my_dir)s/Pictures as the value of my_pictures and %(home_dir)s/lumberjack as the value of my_dir.

```python
class configparser.ExtendedInterpolation
    An alternative handler for interpolation which implements a more
```
advanced syntax, used for instance in zc.buildout. Extended interpolation is using \${section:option} to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the section: part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python>Versions/${Frameworks:Python}
```
Mapping Protocol Access

New in version 3.2.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of `configparser`, the mapping interface implementation is using the `parser['section'][option]` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

`configparser` objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the `MutableMapping` ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner [1]. E.g. for `option in parser['section']` yields only `optionxform`ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key 'a', both expressions return `True`:

  ```python
  "a" in parser["section"]
  "A" in parser["section"]
  ```

- All sections include DEFAULTSECT values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the
section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a KeyError.

- **DEFAULTSECT** cannot be removed from the parser:
  - trying to delete it raises `ValueError`,
  - `parser.clear()` leaves it intact,
  - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.

- `parser.items()` is compatible with the mapping protocol (returns a list of `section_name`, `section_proxy` pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.items(section, raw, vars)`. The latter call returns a list of `option`, `value` pairs for a specified section, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.
Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. `configparser` goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- **defaults**, default value: `None`  
  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don’t specify values which are the same as the documented default.
  
  Hint: if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

- **dict_type**, default value: `collections.OrderedDict`  
  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the default ordered dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back. You can also use a regular dictionary for performance reasons.

  Please note: there are ways to add a set of key-value pairs in a
single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered because dict preserves order from Python 3.7. For example:

```python
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value',
                                 'key2': 'value',
                                 'key3': 'value',
                                 'section2': {'keyA': 'value',
                                                  'keyB': 'value',
                                                  'keyC': 'value',
                                 'section3': {'foo': 'x',
                                              'bar': 'y',
                                              'baz': 'z'}}})
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]  
['foo', 'bar', 'baz']
```

- `allow_no_value`, default value: False

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by `configparser`. The `allow_no_value` parameter to the constructor can be used to indicate that such values should be accepted:

```python
>>> import configparser

>>> sample_config = """
... [mysqld]
...  user = mysql
...  pid-file = /var/run/mysqld/mysqld.pid
...  skip-external-locking
"""
```
old_passwords = 1

skip-bdb

# we don't need ACID today
skip-innodb

""

```python
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config['mysqld']['user']
'mysql'

>>> # Settings without values provide None:
>>> config['mysqld']['skip-bdb']

>>> # Settings which aren't specified still raise an error:
>>> config['mysqld']['does-not-exist']
Traceback (most recent call last):
  ... KeyError: 'does-not-exist'
```

- **delimiters**, default value: `('=', ':')`

Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

See also the **space_around_delimiters** argument to `ConfigParser.write()`.

- **comment_prefixes**, default value: `('#', ';')`

- **inline_comment_prefixes**, default value: `None`
Comment prefixes are strings that indicate the start of a valid comment within a config file. `comment_prefixes` are used only on otherwise empty lines (optionally indented) whereas `inline_comment_prefixes` can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and '####' and ';;' are used as prefixes for whole line comments.

*Changed in version 3.2:* In previous versions of `configparser` behaviour matched `comment_prefixes=('#',';')` and `inline_comment_prefixes=(';','').`

Please note that config parsers don’t support escaping of comment prefixes so using `inline_comment_prefixes` may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting `inline_comment_prefixes`. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```python
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation)
>>> # the default BasicInterpolation could be used
>>> parser.read_string(""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
... ${hash}!/usr/bin/env python
... ${hash} -*- coding: utf-8 -*-
...
... extensions =
... enabled_extension
""
```
strict, default value: True

When set to True, the parser will not allow for any section or option duplicates while reading from a single source (using read_file(), read_string() or read_dict()). It is recommended to use strict parsers in new applications.

Changed in version 3.2: In previous versions of configparser behaviour matched strict=False.
• `empty_lines_in_values`, default value: `True`

In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
    value with a gotcha

this = is still a part of the multiline value of 'key'
```

This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

• `default_section`, default value: `configparser.DEFAULTSECT` (that is: "DEFAULT")

The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally called "DEFAULT" but this can be customized to point to any other valid section name. Some typical values include: "general" or "common". The
name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- **interpolation**, default value: `configparser.BasicInterpolation`
  
  Interpolation behaviour may be customized by providing a custom handler through the `interpolation` argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the dedicated documentation section. `RawConfigParser` has a default value of `None`.

- **converters**, default value: not set

  Config parsers provide option value getters that perform type conversion. By default `getint()`, `getfloat()`, and `getboolean()` are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{ 'decimal': decimal.Decimal }` would add `getdecimal()` on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0).`
If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

**ConfigParser.BOOLEAN_STATES**

By default when using `getboolean()`, config parsers consider the following values `True: '1', 'yes', 'true', 'on'` and the following values `False: '0', 'no', 'false', 'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```python
g>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...  
ValueError: Not a boolean: nope  
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope'}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `accept/reject` or `enabled/disabled`.

**ConfigParser.optionxform(option)**

This method transforms option names on every read, get, or set
operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```python
>>> config = ""
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... 
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

**ConfigParser.** **SECTCRE**

A compiled regular expression used to parse section headers. The default matches `[section]` to the name "section". Whitespace is considered part of the section name, thus `[larch ]` will be read as a section of name "larch ". Override this attribute if that's unsuitable. For example:

```python
>>> import re
>>> config = ""
... [Section 1]
```
... option = value
...
... [  Section 2  ]
... another = val
... 

```python
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r'\[ *(?P<header>[^]\]+?) *\]')
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

**Note:** While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options `allow_no_value` and `delimiters`. 
Legacy API Examples

Mainly because of backwards compatibility concerns, `configparser` provides also a legacy API with explicit `get/set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```python
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set function to assign
# non-string values to keys internally, but will raise an error
# attempting to write to a file or when you get it in a
# values using the mapping protocol or ConfigParser's
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```python
import configparser
```
config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float.
# getint() and getboolean() also do this for their respective types.
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))

To get interpolation, use ConfigParser:

import configparser
cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> 'Python is fun!'
print(cfg.get('Section1', 'foo', raw=True))   # -> '%(bar)s is %(baz)s!'

# The optional *vars* argument is a dict with members that will take precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation', 'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value.
print(cfg.get('Section1', 'foo'))
    # -> "Python is fun!"
Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```python
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life'
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))  # -> "Python is fun!"
print(config.get('Section1', 'monster', fallback=None))  # -> None
print(config.get('Section1', 'foo', fallback='Monty is not.'))  # -> "Python is fun!"

# A bare print(config.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))  # -> "Life is hard!
```
ConfigParser Objects

class configparser.ConfigParser(defaults=None, dict_type=collections.OrderedDict, allow_no_value=False, delimiters=('=', ';'), comment_prefixes=('#', ';'), inline_comment_prefixes=None, strict=True, empty_lines_in_values=True, default_section=configparser.DEFAULTSECT, interpolation=BasicInterpolation(), converters={})

The main configuration parser. When defaults is given, it is initialized into the dictionary of intrinsic defaults. When dict_type is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When delimiters is given, it is used as the set of substrings that divide keys from values. When comment_prefixes is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When inline_comment_prefixes is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When strict is True (the default), the parser won’t allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising DuplicateSectionError or DuplicateOptionError. When empty_lines_in_values is False (default: True), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When allow_no_value is True (default: False), options without values are accepted; the value held for these is None and they are serialized without the trailing delimiter.

When default_section is given, it specifies the name for the special section holding default values for other sections and interpolation
purposes (normally named "DEFAULT"). This value can be retrieved and changed on runtime using the default_section instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the interpolation argument. None can be used to turn off interpolation completely, ExtendedInterpolation() provides a more advanced variant inspired by zc.buildout. More on the subject in the dedicated documentation section.

All option names used in interpolation will be passed through the optionxform() method just like any other option name reference. For example, using the default implementation of optionxform() (which converts option names to lower case), the values foo %(bar)s and foo %(BAR)s are equivalent.

When converters is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding get*() method on the parser object and section proxies.

*Changed in version 3.1:* The default dict_type is collections.OrderedDict.

*Changed in version 3.2:* allow_no_value, delimiters, comment_prefixes, strict, empty_lines_in_values, default_section and interpolation were added.

*Changed in version 3.5:* The converters argument was added.

*Changed in version 3.7:* The defaults argument is read with read_dict(), providing consistent behavior across the parser:
non-string keys and values are implicitly converted to strings.

**defaults()**

Return a dictionary containing the instance-wide defaults.

**sections()**

Return a list of the sections available; the *default section* is not included in the list.

**add_section(section)**

Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised. The name of the section must be a string; if not, *TypeError* is raised.

*Changed in version 3.2:* Non-string section names raise *TypeError.*

**has_section(section)**

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

**options(section)**

Return a list of options available in the specified *section*.

**has_option(section, option)**

If the given *section* exists, and contains the given *option*, return *True*; otherwise return *False*. If the specified *section* is *None* or an empty string, DEFAULT is assumed.

**read(filenames, encoding=None)**

Attempt to read and parse an iterable of filenames, returning a
list of filenames which were successfully parsed.

If *filenames* is a string, a *bytes* object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user’s home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the *ConfigParser* instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using *read_file()* before calling *read()* for any optional files:

```python
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(["site.cfg", os.path.expanduser('~/.myapp.cfg')])
```

*New in version 3.2*: The *encoding* parameter. Previously, all files were read using the default encoding for *open()*.

*New in version 3.6.1*: The *filenames* parameter accepts a *path-like object*.

*New in version 3.7*: The *filenames* parameter accepts a *bytes* object.

*read_file*(f, *source=None*)
Read and parse configuration data from \( f \) which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument \( source \) specifies the name of the file being read. If not given and \( f \) has a name attribute, that is used for \( source \); the default is '<???>'.

*New in version 3.2:* Replaces \( \text{readfp}() \).

**read_string**(\( \text{string}, \text{source}='<\text{string}>' \))

Parse configuration data from a string.

Optional argument \( source \) specifies a context-specific name of the string passed. If not given, '<string>' is used. This should commonly be a filesystem path or a URL.

*New in version 3.2.*

**read_dict**(\( \text{dictionary}, \text{source}='<\text{dict}>' \))

Load configuration from any object that provides a dict-like \( \text{items}() \) method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument \( source \) specifies a context-specific name of the dictionary passed. If not given, <dict> is used.

This method can be used to copy state between parsers.

*New in version 3.2.*
get(section, option, *, raw=False, vars=None[, fallback])

Get an option value for the named section. If vars is provided, it must be a dictionary. The option is looked up in vars (if provided), section, and in DEFAULTSECT in that order. If the key is not found and fallback is provided, it is used as a fallback value. None can be provided as a fallback value.

All the '%%' interpolations are expanded in the return values, unless the raw argument is true. Values for interpolation keys are looked up in the same manner as the option.

Changed in version 3.2: Arguments raw, vars and fallback are keyword only to protect users from trying to use the third argument as the fallback fallback (especially when using the mapping protocol).

getint(section, option, *, raw=False, vars=None[, fallback])

A convenience method which coerces the option in the specified section to an integer. See get() for explanation of raw, vars and fallback.

getfloat(section, option, *, raw=False, vars=None[, fallback])

A convenience method which coerces the option in the specified section to a floating point number. See get() for explanation of raw, vars and fallback.

getboolean(section, option, *, raw=False, vars=None[, fallback])

A convenience method which coerces the option in the specified section to a Boolean value. Note that the accepted values for the option are '1', 'yes', 'true', and 'on',
which cause this method to return `True`, and `'0'`, `'no'`, 'false', and 'off', which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `ValueError`. See `get()` for explanation of `raw`, `vars` and `fallback`.

`items(raw=False, vars=None)`

`items(section, raw=False, vars=None)`

When `section` is not given, return a list of `section_name`, `section_proxy` pairs, including DEFAULTSECT.

Otherwise, return a list of `name, value` pairs for the options in the given `section`. Optional arguments have the same meaning as for the `get()` method.

`set(section, option, value)`

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. `option` and `value` must be strings; if not, `TypeError` is raised.

`write(fileobject, space_around_delimiters=True)`

Write a representation of the configuration to the specified `file object`, which must be opened in text mode (accepting strings). This representation can be parsed by a future `read()` call. If `space_around_delimiters` is true, delimiters between keys and values are surrounded by spaces.

`remove_option(section, option)`

Remove the specified `option` from the specified `section`. If the section does not exist, raise `NoSectionError`. If the option existed to be removed, return `True`; otherwise return `False`. 
remove_section(section)

Remove the specified section from the configuration. If the section in fact existed, return True. Otherwise return False.

optionxform(option)

Transforms the option name option as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of option; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don’t need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to str, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before optionxform() is called.

readfp(fp, filename=None)

Deprecated since version 3.2: Use read_file() instead.

Changed in version 3.2: readfp() now iterates on fp instead of calling fp.readline().

For existing code calling readfp() with arguments which don’t support iteration, the following generator may be used as a wrapper around the file-like object:
```python
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

Instead of `parser.readfp(fp)` use
`parser.read_file(readline_generator(fp))`.

`configparser.MAX_INTERPOLATION_DEPTH`

The maximum depth for recursive interpolation for `get()` when the `raw` parameter is false. This is relevant only when the default `interpolation` is used.
RawConfigParser Objects

class configparser.RawConfigParser (defaults=None, dict_type=collections.OrderedDict, allow_no_value=False, *, delimiters=('=', ':'), comment_prefixes=('#', ';'), inline_comment_prefixes=None, strict=True, empty_lines_in_values=True, default_section=configparser.DEFAULTSECT[, interpolation])

Legacy variant of the ConfigParser. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe add_section and set methods, as well as the legacy defaults= keyword argument handling.

**Note:** Consider using ConfigParser instead which checks types of the values to be stored internally. If you don’t want interpolation, you can use ConfigParser(interpolation=None).

**add_section** *(section)*

Add a section named *section* to the instance. If a section by the given name already exists, DuplicateSectionError is raised. If the *default section* name is passed, ValueError is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

**set** *(section, option, value)*

If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. While it is possible
to use RawConfigParser (or ConfigParser with raw parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. Use the mapping protocol API which does not allow such assignments to take place.
Exceptions

exception configparser.Error
    Base class for all other configparser exceptions.

exception configparser.NoSectionError
    Exception raised when a specified section is not found.

exception configparser.DuplicateSectionError
    Exception raised if add_section() is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

    New in version 3.2: Optional source and lineno attributes and arguments to __init__() were added.

exception configparser.DuplicateOptionError
    Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.NoOptionError
    Exception raised when a specified option is not found in the specified section.

exception configparser.InterpolationError
    Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.InterpolationDepthError
Exception raised when string interpolation cannot be completed because the number of iterations exceeds `MAX_INTERPOLATION_DEPTH`. Subclass of `InterpolationError`.

```python
exception configparser.InterpolationMissingOptionError
```

Exception raised when an option referenced from a value does not exist. Subclass of `InterpolationError`.

```python
exception configparser.InterpolationSyntaxError
```

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of `InterpolationError`.

```python
exception configparser.MissingSectionHeaderError
```

Exception raised when attempting to parse a file which has no section headers.

```python
exception configparser.ParseError
```

Exception raised when errors occur attempting to parse a file.

*Changed in version 3.2:* The `filename` attribute and `__init__()` argument were renamed to `source` for consistency.

**Footnotes**

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the Customizing Parser Behaviour section.
The `netrc` class parses and encapsulates the netrc file format used by the Unix `ftp` program and other FTP clients.

```python
class netrc.netrc([file])
```

A `netrc` instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user’s home directory – as determined by `os.path.expanduser()` – will be read. Otherwise, a `FileNotFoundException` exception will be raised. Parse errors will raise `NetrcParseError` with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a `NetrcParseError` if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of `ftp` and other programs that use `.netrc`.

*Changed in version 3.4:* Added the POSIX permission check.

*Changed in version 3.7:* `os.path.expanduser()` is used to find the location of the `.netrc` file when `file` is not passed as argument.

`exception netrc.NetrcParseError`
Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes: `msg` is a textual explanation of the error, `filename` is the name of the source file, and `lineno` gives the line number on which the error was found.
netrc Objects

A netrc instance has the following methods:

netrc.authenticators(host)
   Return a 3-tuple (login, account, password) of authenticators for host. If the netrc file did not contain an entry for the given host, return the tuple associated with the ‘default’ entry. If neither matching host nor default entry is available, return None.

netrc.__repr__()
   Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of netrc have public instance variables:

netrc.hosts
   Dictionary mapping host names to (login, account, password) tuples. The ‘default’ entry, if any, is represented as a pseudo-host by that name.

netrc.macros
   Dictionary mapping macro names to string lists.

Note: Passwords are limited to a subset of the ASCII character set. All ASCII punctuation is allowed in passwords, however, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.
The `xdrlib` module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The `xdrlib` module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

```python
class xdrlib.Packer
    Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

class xdrlib.Unpacker(data)
    Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as data.
```

See also:

**RFC 1014 - XDR: External Data Representation Standard**
This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

**RFC 1832 - XDR: External Data Representation Standard**
Newer RFC that provides a revised definition of XDR.
Packer Objects

Packer instances have the following methods:

Packer. **get_buffer()**

    Returns the current pack buffer as a string.

Packer. **reset()**

    Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate pack_type() method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: pack_uint(), pack_int(), pack_enum(), pack_enum(), pack_bool(), pack_uhyper(), and pack_hyper().

Packer. **pack_float(value)**

    Packs the single-precision floating point number value.

Packer. **pack_double(value)**

    Packs the double-precision floating point number value.

The following methods support packing strings, bytes, and opaque data:

Packer. **pack_fstring(n, s)**

    Packs a fixed length string, s. n is the length of the string but it is not packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

Packer. **pack_fopaque(n, data)**
Packs a fixed length opaque data stream, similarly to `pack_fstring()`.

**Packer**. **pack_string**(*s*)

Packs a variable length string, *s*. The length of the string is first packed as an unsigned integer, then the string data is packed with `pack_fstring()`.

**Packer**. **pack_opaque**(*data*)

Packs a variable length opaque data string, similarly to `pack_string()`.

**Packer**. **pack_bytes**(*bytes*)

Packs a variable length byte stream, similarly to `pack_string()`.

The following methods support packing arrays and lists:

**Packer**. **pack_list**(*list, pack_item*)

Packs a *list* of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. `pack_item` is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

For example, to pack a list of integers, the code might appear like this:

```python
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

**Packer**. **pack_farray**(*n, array, pack_item*)
Packs a fixed length list (*array*) of homogeneous items. *n* is the length of the list; it is *not* packed into the buffer, but a `ValueError` exception is raised if `len(array)` is not equal to *n*. As above, `pack_item` is the function used to pack each element.

Packer. **pack_array**(list, pack_item)

Packs a variable length *list* of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in `pack_farray()` above.
Unpacker Objects

The **Unpacker** class offers the following methods:

**Unpacker.** `reset(data)`
- Resets the string buffer with the given `data`.

**Unpacker.** `get_position()`
- Returns the current unpack position in the data buffer.

**Unpacker.** `set_position(position)`
- Sets the data buffer unpack position to `position`. You should be careful about using `get_position()` and `set_position()`.

**Unpacker.** `get_buffer()`
- Returns the current unpack data buffer as a string.

**Unpacker.** `done()`
- Indicates unpack completion. Raises an **Error** exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a **Packer**, can be unpacked with an **Unpacker**. Unpacking methods are of the form `unpack_type()`, and take no arguments. They return the unpacked object.

**Unpacker.** `unpack_float()`
- Unpacks a single-precision floating point number.

**Unpacker.** `unpack_double()`
- Unpacks a double-precision floating point number, similarly to `unpack_float()`.
In addition, the following methods unpack strings, bytes, and opaque data:

Unpacker. **unpack_fstring**(\(n\))

Unpacks and returns a fixed length string. \(n\) is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

Unpacker. **unpack_fopaque**(\(n\))

Unpacks and returns a fixed length opaque data stream, similarly to `unpack_fstring()`.

Unpacker. **unpack_string**()

Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with `unpack_fstring()`.

Unpacker. **unpack_opaque**()

Unpacks and returns a variable length opaque data string, similarly to `unpack_string()`.

Unpacker. **unpack_bytes**()

Unpacks and returns a variable length byte stream, similarly to `unpack_string()`.

The following methods support unpacking arrays and lists:

Unpacker. **unpack_list**(\(\text{unpack\_item}\))

Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. \(\text{unpack\_item}\) is the function that is called to unpack the items.
Unpacker. **unpack_farray**(*n*, *unpack_item*)

Unpacks and returns (as a list) a fixed length array of homogeneous items. *n* is number of list elements to expect in the buffer. As above, *unpack_item* is the function used to unpack each element.

Unpacker. **unpack_array**(*unpack_item*)

Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in *unpack_farray()* above.
Exceptions

Exceptions in this module are coded as class instances:

`exception xdrlib.Error`  
The base exception class. `Error` has a single public attribute `msg` containing the description of the error.

`exception xdrlib.ConversionError`  
Class derived from `Error`. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:

```python
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```
plistlib — Generate and parse Mac OS X .plist files

Source code: Lib/plistlib.py

This module provides an interface for reading and writing the “property list” files used mainly by Mac OS X and supports both binary and XML plist files.

The property list (.plist) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the dump() and load() functions.

To work with plist data in bytes objects, use dumps() and loads().

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), Data, bytes, bytesarray or datetime.datetime objects.

Changed in version 3.4: New API, old API deprecated. Support for binary format plists added.

See also:

PList manual page
   Apple’s documentation of the file format.
This module defines the following functions:

```python
plistlib.load(fp, *, fmt=None, use_builtin_types=True, dict_type=dict)
```

Read a plist file. `fp` should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The `fmt` is the format of the file and the following values are valid:

- `None`: Autodetect the file format
- `FMT_XML`: XML file format
- `FMT_BINARY`: Binary plist format

If `use_builtin_types` is true (the default) binary data will be returned as instances of `bytes`, otherwise it is returned as instances of `Data`.

The `dict_type` is the type used for dictionaries that are read from the plist file.

XML data for the `FMT_XML` format is parsed using the Expat parser from `xml.parsers.expat` – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser for the binary format raises `InvalidFileException` when the file cannot be parsed.

*New in version 3.4.*

```python
plistlib.loads(data, *, fmt=None, use_builtin_types=True, dict_type=dict)
```

Load a plist from a bytes object. See `load()` for an explanation of the keyword arguments.
New in version 3.4.

`plistlib.dump(value, fp, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Write `value` to a plist file. `fp` should be a writable, binary file object.

The `fmt` argument specifies the format of the plist file and can be one of the following values:

- **FMT_XML**: XML formatted plist file
- **FMT_BINARY**: Binary formatted plist file

When `sort_keys` is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When `skipkeys` is false (the default) the function raises `TypeError` when a key of a dictionary is not a string, otherwise such keys are skipped.

A `TypeError` will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An `OverflowError` will be raised for integer values that cannot be represented in (binary) plist files.

New in version 3.4.

`plistlib.dumps(value, *, fmt=FMT_XML, sort_keys=True, skipkeys=False)`

Return `value` as a plist-formatted bytes object. See the documentation for `dump()` for an explanation of the keyword arguments of this function.
New in version 3.4.

The following functions are deprecated:

**plistlib.** **readPlist**(pathOrFile)

Read a plist file. pathOrFile may be either a file name or a (readable and binary) file object. Returns the unpacked root object (which usually is a dictionary).

This function calls **load()** to do the actual work, see the documentation of that function for an explanation of the keyword arguments.

*Deprecated since version 3.4:* Use **load()** instead.

*Changed in version 3.7:* Dict values in the result are now normal dicts. You no longer can use attribute access to access items of these dictionaries.

**plistlib.** **writePlist**(rootObject, pathOrFile)

Write rootObject to an XML plist file. pathOrFile may be either a file name or a (writable and binary) file object

*Deprecated since version 3.4:* Use **dump()** instead.

**plistlib.** **readPlistFromBytes**(data)

Read a plist data from a bytes object. Return the root object.

See **load()** for a description of the keyword arguments.

*Deprecated since version 3.4:* Use **loads()** instead.

*Changed in version 3.7:* Dict values in the result are now normal dicts. You no longer can use attribute access to access items of
these dictionaries.

`plistlib.writePlistToBytes(rootObject)`  
Return `rootObject` as an XML plist-formatted bytes object.

*Deprecated since version 3.4*: Use `dumps()` instead.

The following classes are available:

`class plistlib.Data(data)`  
Return a “data” wrapper object around the bytes object `data`. This is used in functions converting from/to plists to represent the `<data>` type available in plists.

It has one attribute, `data`, that can be used to retrieve the Python bytes object stored in it.

*Deprecated since version 3.4*: Use a `bytes` object instead.

The following constants are available:

`plistlib.FMT_XML`  
The XML format for plist files.

*New in version 3.4.*

`plistlib.FMT_BINARY`  
The binary format for plist files

*New in version 3.4.*
Examples

Generating a plist:

```python
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime()),
)

with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

Parsing a plist:

```python

with open(fileName, 'rb') as fp:
    pl = load(fp)

print(pl["aKey"])
```
Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. On Unix systems, the `crypt` module may also be available. Here’s an overview:

- `hashlib` — Secure hashes and message digests
  - Hash algorithms
  - SHAKE variable length digests
  - Key derivation
  - BLAKE2
    - Creating hash objects
    - Constants
    - Examples
      - Simple hashing
      - Using different digest sizes
      - Keyed hashing
      - Randomized hashing
      - Personalization
      - Tree mode
  - Credits
- `hmac` — Keyed-Hashing for Message Authentication
- `secrets` — Generate secure random numbers for managing secrets
  - Random numbers
  - Generating tokens
    - How many bytes should tokens use?
  - Other functions
  - Recipes and best practices
hashlib — Secure hashes and message digests

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA’s MD5 algorithm (defined in Internet RFC 1321). The terms “secure hash” and “message digest” are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

Note: If you want the adler32 or crc32 hash functions, they are available in the zlib module.

Warning: Some algorithms have known hash collision weaknesses, refer to the “See also” section at the end.
Hash algorithms

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use sha256() to create a SHA-256 hash object. You can now feed this object with bytes-like objects (normally bytes) using the update() method. At any point you can ask it for the digest of the concatenation of the data fed to it so far using the digest() or hexdigest() methods.

**Note:** For better multithreading performance, the Python GIL is released for data larger than 2047 bytes at object creation or on update.

**Note:** Feeding string objects into update() is not supported, as hashes work on bytes, not on characters.

Constructors for hash algorithms that are always present in this module are sha1(), sha224(), sha256(), sha384(), sha512(), blake2b(), and blake2s(). md5() is normally available as well, though it may be missing if you are using a rare “FIPS compliant” build of Python. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform. On most platforms the sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256() are also available.

*New in version 3.6:* SHA3 (Keccak) and SHAKE constructors sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256().

*New in version 3.6:* blake2b() and blake2s() were added.
For example, to obtain the digest of the byte string b'Nobody inspects the spammish repetition':

```python
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\x00o\xa5u+7\xfd\x8fN\x84:\xa6\xaf\x0c\x95\x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

More condensed:

```python
>>> hashlib.sha224(b"Nobody inspects the spammish repetition")
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e'
```

`hashlib.new(name[, data])`

Is a generic constructor that takes the string `name` of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than `new()` and should be preferred.

Using `new()` with an algorithm provided by OpenSSL:

```python
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894ecc'
```
Hashlib provides the following constant attributes:

**hashlib.algorithms_guaranteed**

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms. Note that ‘md5’ is in this list despite some upstream vendors offering an odd “FIPS compliant” Python build that excludes it.

*New in version 3.2.*

**hashlib.algorithms_available**

A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to `new()`. `algorithms_guaranteed` will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

*New in version 3.2.*

The following values are provided as constant attributes of the hash objects returned by the constructors:

**hash.digest_size**

The size of the resulting hash in bytes.

**hash.block_size**

The internal block size of the hash algorithm in bytes.

A hash object has the following attributes:

**hash.name**

The canonical name of this hash, always lowercase and always suitable as a parameter to `new()` to create another hash of this type.
Changed in version 3.4: The name attribute has been present in CPython since its inception, but until Python 3.4 was not formally specified, so may not exist on some platforms.

A hash object has the following methods:

**hash**. **update**(data)

Update the hash object with the bytes-like object. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

Changed in version 3.1: The Python GIL is released to allow other threads to run while hash updates on data larger than 2047 bytes is taking place when using hash algorithms supplied by OpenSSL.

**hash**. **digest**()

Return the digest of the data passed to the `update()` method so far. This is a bytes object of size `digest_size` which may contain bytes in the whole range from 0 to 255.

**hash**. **hexdigest**()

Like `digest()` except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**hash**. **copy**()

Return a copy (“clone”) of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.
SHAKE variable length digests

The `shake_128()` and `shake_256()` algorithms provide variable length digests with `length_in_bits//2` up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

`shake.digest(length)`
Return the digest of the data passed to the `update()` method so far. This is a bytes object of size `length` which may contain bytes in the whole range from 0 to 255.

`shake.hexdigest(length)`
Like `digest()` except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.
Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as \texttt{sha1(password)} are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a \texttt{salt}.

\texttt{hashlib.pbkdf2\_hmac(hash\_name, password, salt, iterations, dklen=None)}

The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudorandom function.

The string \texttt{hash\_name} is the desired name of the hash digest algorithm for HMAC, e.g. ‘sha1’ or ‘sha256’. \texttt{password} and \texttt{salt} are interpreted as buffers of bytes. Applications and libraries should limit \texttt{password} to a sensible length (e.g. 1024). \texttt{salt} should be about 16 or more bytes from a proper source, e.g. \texttt{os.urandom()}.  

The number of \texttt{iterations} should be chosen based on the hash algorithm and computing power. As of 2013, at least 100,000 iterations of SHA-256 are suggested.

\texttt{dklen} is the length of the derived key. If \texttt{dklen} is \texttt{None} then the digest size of the hash algorithm \texttt{hash\_name} is used, e.g. 64 for SHA-512.

```python
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', iterations=100000, dklen=64)
>>> binascii.hexlify(dk)
```

b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0f1234567890abcdefg'
New in version 3.4.

**Note:** A fast implementation of `pbkdf2_hmac` is available with OpenSSL. The Python implementation uses an inline version of `hmac`. It is about three times slower and doesn’t release the GIL.

The function provides scrypt password-based key derivation function as defined in [RFC 7914](https://tools.ietf.org/html/rfc7914).

`password` and `salt` must be bytes-like objects. Applications and libraries should limit `password` to a sensible length (e.g. 1024). `salt` should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

`n` is the CPU/Memory cost factor, `r` the block size, `p` parallelization factor and `maxmem` limits memory (OpenSSL 1.1.0 defaults to 32 MiB). `dklen` is the length of the derived key.

**Availability:** OpenSSL 1.1+.

New in version 3.6.
BLAKE2

BLAKE2 is a cryptographic hash function defined in RFC 7693 that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC), **salted hashing**, **personalization**, and **tree hashing**.

Hash objects from this module follow the API of standard library’s `hashlib` objects.

Creating hash objects

New hash objects are created by calling constructor functions:

```python
hashlib.blake2b(data=b'', *, digest_size=64, key=b'', salt=b'',
                 person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0,
                 node_depth=0, inner_size=0, last_node=False)
```

```python
hashlib.blake2s(data=b'', *, digest_size=32, key=b'', salt=b'',
                 person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0,
                 node_depth=0, inner_size=0, last_node=False)
```

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- **data**: initial chunk of data to hash, which must be bytes-like object.
It can be passed only as positional argument.

- **digest_size**: size of output digest in bytes.
- **key**: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- **salt**: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- **person**: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

The following table shows limits for general parameters (in bytes):

<table>
<thead>
<tr>
<th>Hash</th>
<th>digest_size</th>
<th>len(key)</th>
<th>len(salt)</th>
<th>len(person)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE2b</td>
<td>64</td>
<td>64</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>BLAKE2s</td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**Note:** BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, b'salt' and b'salt\x00' is the same value. (This is not the case for key.)

These sizes are available as module **constants** described below.

Constructor functions also accept the following tree hashing parameters:

- **fanout**: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- **depth**: maximal depth of tree (1 to 255, 255 if unlimited, 1 in sequential mode).
- **leaf_size**: maximal byte length of leaf (0 to 2**32-1, 0 if unlimited or in sequential mode).
- **node_offset**: node offset (0 to 2**64-1 for BLAKE2b, 0 to 2**48-1...
for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- `node_depth`: node depth (0 to 255, 0 for leaves, or in sequential mode).
- `inner_size`: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- `last_node`: boolean indicating whether the processed node is the last one (`False` for sequential mode).

See section 2.10 in [BLAKE2 specification](#) for comprehensive review of tree hashing.

**Constants**

`blake2b.SALT_SIZE`

`blake2s.SALT_SIZE`

Salt length (maximum length accepted by constructors).

`blake2b.PERSON_SIZE`
`blake2s.PERSON_SIZE`

Personalization string length (maximum length accepted by constructors).

`blake2b.MAX_KEY_SIZE`

`blake2s.MAX_KEY_SIZE`

Maximum key size.

`blake2b.MAX_DIGEST_SIZE`

`blake2s.MAX_DIGEST_SIZE`

Maximum digest size that the hash function can output.

**Examples**

**Simple hashing**

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (`blake2b()` or `blake2s()`), then update it with the data by calling `update()` on the object, and, finally, get the digest out of the object by calling `digest()` (or `hexdigest()` for hex-encoded string).

```python
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d1289743a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb0aa1bb0388bbeac29183'
```

As a shortcut, you can pass the first chunk of data to update directly to
the constructor as the positional argument:

```python
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974'
```

You can call `hash.update()` as many times as you need to iteratively update the hash:

```python
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...    h.update(item)
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974'
```

Using different digest sizes

BLAKE2 has configurable size of digests up to 64 bytes for BLAKE2b and up to 32 bytes for BLAKE2s. For example, to replace SHA-1 with BLAKE2b without changing the size of output, we can tell BLAKE2b to produce 20-byte digests:

```python
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function'
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```
Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```python
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

**Keyed hashing**

Keyed hashing can be used for authentication as a faster and simpler replacement for *Hash-based message authentication code* (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```python
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=10)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign
cookies sent to users and later verify them to make sure they weren’t tampered with:

```python
>>> from hashlib import blake2b
>>> from hmac import compare_digest

>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16

>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')

>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)

>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("\{0},\{1}\".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Even though there’s a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with `hmac` module:

```python
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
```
Randomized hashing

By setting salt parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.

(NIST SP-800-106 “Randomized Hashing for Digital Signatures”)
In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

**Warning:** *Salted hashing* (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See [BLAKE2 FAQ](#) for more information.

```python
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

**Personalization**

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

```
We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely
```
different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.

(The Skein Hash Function Family, p. 21)

BLAKE2 can be personalized by passing bytes to the `person` argument:

```python
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9ea52264a80b75005e65619'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```python
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNf2
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt')
>>> mac_key = blake2s(key=orig_key, person=b'kMAC')
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuHB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWP1YK1e/nWfu0WSEb0KRCjhDeP/o=
```
Tree mode

Here’s an example of hashing a minimal tree with two leaf nodes:

```
10
/ \
00 01
```

This example uses 64-byte internal digests, and returns the 32-byte final digest:

```python
>>> from hashlib import blake2b
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64

>>> buf = bytearray(6000)

>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=0, node_depth=0, last_node=False)                

>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=1, node_depth=0, last_node=False)                

>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=0, node_depth=1, last_node=False)                

>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
't3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```
Credits

**BLAKE2** was designed by Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein based on SHA-3 finalist **BLAKE** created by Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan.

It uses core algorithm from **ChaCha** cipher designed by Daniel J. Bernstein.

The stdlib implementation is based on **pyblake2** module. It was written by Dmitry Chestnykh based on C implementation written by Samuel Neves. The documentation was copied from **pyblake2** and written by Dmitry Chestnykh.

The C code was partly rewritten for Python by Christian Heimes.

The following public domain dedication applies for both C hash function implementation, extension code, and this documentation:

To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.

You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see [https://creativecommons.org/publicdomain/zero/1.0/](https://creativecommons.org/publicdomain/zero/1.0/).

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:
See also:

**Module hmac**
A module to generate message authentication codes using hashes.

**Module base64**
Another way to encode binary hashes for non-binary environments.

[https://blake2.net](https://blake2.net)
Official BLAKE2 website.

The FIPS 180-2 publication on Secure Hash Algorithms.

Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

[https://www.ietf.org/rfc/rfc2898.txt](https://www.ietf.org/rfc/rfc2898.txt)
PKCS #5: Password-Based Cryptography Specification Version 2.0
hmac — Keyed-Hashing for Message Authentication

Source code: Lib/hmac.py

This module implements the HMAC algorithm as described by RFC 2104.

```
hmac.new(key, msg=None, digestmod=None)
```

Return a new hmac object. `key` is a bytes or bytearray object giving the secret key. If `msg` is present, the method call `update(msg)` is made. `digestmod` is the digest name, digest constructor or module for the HMAC object to use. It supports any name suitable to `hashlib.new()` and defaults to the `hashlib.md5` constructor.

*Changed in version 3.4:* Parameter `key` can be a bytes or bytearray object. Parameter `msg` can be of any type supported by `hashlib`. Parameter `digestmod` can be the name of a hash algorithm.

*Deprecated since version 3.4, will be removed in version 3.8:* MD5 as implicit default digest for `digestmod` is deprecated.

```
hmac.digest(key, msg, digest)
```

Return digest of `msg` for given secret `key` and `digest`. The function is equivalent to `HMAC(key, msg, digest).digest()`, but uses an optimized C or inline implementation, which is faster for messages that fit into memory. The parameters `key`, `msg`, and `digest` have the same meaning as in `new()`.

CPython implementation detail, the optimized C implementation is
only used when *digest* is a string and name of a digest algorithm, which is supported by OpenSSL.

*New in version 3.7.*

An HMAC object has the following methods:

**HMAC**. **update**(*msg*)

Update the hmac object with *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

*Changed in version 3.4:* Parameter *msg* can be of any type supported by *hashlib*.

**HMAC**. **digest**()

Return the digest of the bytes passed to the *update()* method so far. This bytes object will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

**Warning:** When comparing the output of *digest()* to an externally-supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC**. **hexdigest**()

Like *digest()* except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.
Warning: When comparing the output of `hexdigest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC.** `copy()`
Return a copy ("clone") of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

A hash object has the following attributes:

**HMAC.** `digest_size`
The size of the resulting HMAC digest in bytes.

**HMAC.** `block_size`
The internal block size of the hash algorithm in bytes.

New in version 3.4.

**HMAC.** `name`
The canonical name of this HMAC, always lowercase, e.g. `hmac-md5`.

New in version 3.4.

This module also provides the following helper function:

**hmac.** `compare_digest(a, b)`
Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. `a` and `b` must both be of the same type: either `str` (ASCII only, as e.g. returned
by `HMAC.hexdigest()`), or a `bytes-like object`.

**Note:** If `a` and `b` are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of `a` and `b`—but not their values.

*New in version 3.3.*

**See also:**

**Module** [hashlib](https://docs.python.org/3/library/hashlib.html)

The Python module providing secure hash functions.
secrets — Generate secure random numbers for managing secrets

New in version 3.6.

Source code: Lib/secrets.py

The secrets module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.

In particularly, secrets should be used in preference to the default pseudo-random number generator in the random module, which is designed for modelling and simulation, not security or cryptography.

See also: PEP 506
Random numbers

The secrets module provides access to the most secure source of randomness that your operating system provides.

class secrets.SystemRandom
    A class for generating random numbers using the highest-quality sources provided by the operating system. See random.SystemRandom for additional details.

secrets.choice(sequence)
    Return a randomly-chosen element from a non-empty sequence.

secrets.randbelow(n)
    Return a random int in the range [0, n).

secrets.randbits(k)
    Return an int with k random bits.
Generating tokens

The `secrets` module provides functions for generating secure tokens, suitable for applications such as password resets, hard-to-guess URLs, and similar.

`secrets.token_bytes([nbytes=None])`

Return a random byte string containing `nbytes` number of bytes. If `nbytes` is `None` or not supplied, a reasonable default is used.

```python
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

`secrets.token_hex([nbytes=None])`

Return a random text string, in hexadecimal. The string has `nbytes` random bytes, each byte converted to two hex digits. If `nbytes` is `None` or not supplied, a reasonable default is used.

```python
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

`secrets.token_urlsafe([nbytes=None])`

Return a random URL-safe text string, containing `nbytes` random bytes. The text is Base64 encoded, so on average each byte results in approximately 1.3 characters. If `nbytes` is `None` or not supplied, a reasonable default is used.

```python
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

How many bytes should tokens use?
To be secure against brute-force attacks, tokens need to have sufficient randomness. Unfortunately, what is considered sufficient will necessarily increase as computers get more powerful and able to make more guesses in a shorter period. As of 2015, it is believed that 32 bytes (256 bits) of randomness is sufficient for the typical use-case expected for the secrets module.

For those who want to manage their own token length, you can explicitly specify how much randomness is used for tokens by giving an `int` argument to the various `token_*` functions. That argument is taken as the number of bytes of randomness to use.

Otherwise, if no argument is provided, or if the argument is None, the `token_*` functions will use a reasonable default instead.

**Note:** That default is subject to change at any time, including during maintenance releases.
Other functions

secrets.compare_digest(a, b)

Return True if strings a and b are equal, otherwise False, in such a way as to reduce the risk of timing attacks. See hmac.compare_digest() for additional details.
Recipes and best practices

This section shows recipes and best practices for using secrets to manage a basic level of security.

Generate an eight-character alphanumeric password:

```python
import string
alphabet = string.ascii_letters + string.digits
password = ''.join(choice(alphabet) for i in range(8))
```

**Note:** Applications should not store passwords in a recoverable format, whether plain text or encrypted. They should be salted and hashed using a cryptographically-strong one-way (irreversible) hash function.

Generate a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits:

```python
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password) and any(c.isupper() for c in password) and sum(c.isdigit() for c in password) >= 3):
        break
```

Generate an **XKCD-style passphrase**:

```bash
# On standard Linux systems, use a convenient dictionary
```
# Other platforms may need to provide their own word-list.

```python
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for _ in range ...)
```

Generate a hard-to-guess temporary URL containing a security token suitable for password recovery applications:

```python
url = 'https://mydomain.com/reset=' + token_urlsafe( ...)
```
Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here’s an overview:

- **os** — Miscellaneous operating system interfaces
  - File Names, Command Line Arguments, and Environment Variables
  - Process Parameters
  - File Object Creation
  - File Descriptor Operations
    - Querying the size of a terminal
    - Inheritance of File Descriptors
  - Files and Directories
    - Linux extended attributes
  - Process Management
  - Interface to the scheduler
  - Miscellaneous System Information
  - Random numbers

- **io** — Core tools for working with streams
  - Overview
    - Text I/O
    - Binary I/O
    - Raw I/O
  - High-level Module Interface
    - In-memory streams
  - Class hierarchy
- I/O Base Classes
- Raw File I/O
- Buffered Streams
- Text I/O
  - Performance
    - Binary I/O
    - Text I/O
    - Multi-threading
    - Reentrancy
- time — Time access and conversions
  - Functions
  - Clock ID Constants
  - Timezone Constants
- argparse — Parser for command-line options, arguments and sub-commands
  - Example
    - Creating a parser
    - Adding arguments
    - Parsing arguments
  - ArgumentParser objects
    - prog
    - usage
    - description
    - epilog
    - parents
    - formatter_class
    - prefix_chars
    - fromfile_prefix_chars
    - argument_default
    - allow_abbrev
    - conflict_handler
    - add_help
  - The add_argument() method
    - name or flags
- action
- nargs
- const
- default
- type
- choices
- required
- help
- metavar
- dest
- Action classes
  - The parse_args() method
    - Option value syntax
    - Invalid arguments
    - Arguments containing -
    - Argument abbreviations (prefix matching)
    - Beyond sys.argv
    - The Namespace object
  - Other utilities
    - Sub-commands
    - FileType objects
    - Argument groups
    - Mutual exclusion
    - Parser defaults
    - Printing help
    - Partial parsing
    - Customizing file parsing
    - Exiting methods
    - Intermixed parsing
  - Upgrading optparse code
- getopt — C-style parser for command line options
- logging — Logging facility for Python
  - Logger Objects
  - Logging Levels
- Handler Objects
- Formatter Objects
- Filter Objects
- LogRecord Objects
- LogRecord attributes
- LoggerAdapter Objects
- Thread Safety
- Module-Level Functions
- Module-Level Attributes
- Integration with the warnings module
  - `logging.config` — Logging configuration
    - Configuration functions
    - Configuration dictionary schema
      - Dictionary Schema Details
      - Incremental Configuration
      - Object connections
      - User-defined objects
      - Access to external objects
      - Access to internal objects
      - Import resolution and custom importers
    - Configuration file format
  - `logging.handlers` — Logging handlers
    - StreamHandler
    - FileHandler
    - NullHandler
    - WatchedFileHandler
    - BaseRotatingHandler
    - RotatingFileHandler
    - TimedRotatingFileHandler
    - SocketHandler
    - DatagramHandler
    - SysLogHandler
    - NTEventLogHandler
    - SMTPHandler
- MemoryHandler
- HTTPHandler
- QueueHandler
- QueueListener
- `getpass` — Portable password input
- `curses` — Terminal handling for character-cell displays
  - Functions
  - Window Objects
  - Constants
- `curses.textpad` — Text input widget for curses programs
  - Textbox objects
- `curses.ascii` — Utilities for ASCII characters
- `curses.panel` — A panel stack extension for curses
  - Functions
  - Panel Objects
- `platform` — Access to underlying platform’s identifying data
  - Cross Platform
  - Java Platform
  - Windows Platform
    - Win95/98 specific
  - Mac OS Platform
  - Unix Platforms
- `errno` — Standard errno system symbols
- `ctypes` — A foreign function library for Python
  - `ctypes` tutorial
    - Loading dynamic link libraries
    - Accessing functions from loaded dlls
    - Calling functions
    - Fundamental data types
    - Calling functions, continued
    - Calling functions with your own custom data types
    - Specifying the required argument types (function prototypes)
    - Return types
- Passing pointers (or: passing parameters by reference)
- Structures and unions
- Structure/union alignment and byte order
- Bit fields in structures and unions
- Arrays
- Pointers
- Type conversions
- Incomplete Types
- Callback functions
- Accessing values exported from dlls
- Surprises
- Variable-sized data types
  - ctypes reference
    - Finding shared libraries
    - Loading shared libraries
    - Foreign functions
    - Function prototypes
    - Utility functions
    - Data types
    - Fundamental data types
    - Structured data types
    - Arrays and pointers
**os — Miscellaneous operating system interfaces**

**Source code:** Lib/os.py

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about `path` in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.

**Note:** All functions in this module raise `OSError` in the case of invalid or inaccessible file names and paths, or other arguments that
have the correct type, but are not accepted by the operating system.

```
exception os.error
    An alias for the built-in OSError exception.
```

```
os.name
    The name of the operating system dependent module imported. The following names have currently been registered: 'posix', 'nt', 'java'.
```

**See also:** sys.platform has a finer granularity. os.uname() gives system-dependent version information. The platform module provides detailed checks for the system's identity.
File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the file system encoding to perform this conversion (see `sys.getfilesystemencoding()`).

*Changed in version 3.1:* On some systems, conversion using the file system encoding may fail. In this case, Python uses the *surrogateescape encoding error handler*, which means that undecodable bytes are replaced by a Unicode character U+DCxx on decoding, and these are again translated to the original byte on encoding.

The file system encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions may raise UnicodeErrors.
Process Parameters

These functions and data items provide information and operate on the current process and user.

**os.ctermid()**

Return the filename corresponding to the controlling terminal of the process.

*Availability*: Unix.

**os.environ**

A *mapping* object representing the string environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the `os` module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in `os.environ`, except for changes made by modifying `os.environ` directly.

If the platform supports the `putenv()` function, this mapping may be used to modify the environment as well as query the environment. `putenv()` will be called automatically when the mapping is modified.

On Unix, keys and values use `sys.getfilesystemencoding()` and 'surrogateescape' error handler. Use `environb` if you would like to use a different encoding.
Note: Calling `putenv()` directly does not change `os.environ`, so it's better to modify `os.environ`.

Note: On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

If `putenv()` is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

If the platform supports the `unsetenv()` function, you can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`, and when one of the `pop()` or `clear()` methods is called.

**os.environb**
Bytes version of `environ`: a `mapping` object representing the environment as byte strings. `environ` and `environb` are synchronized (modify `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is True.

*New in version 3.2.*

**os.chdir**(path)
**os.fchdir**(fd)
**os.getcwd()**
These functions are described in *Files and Directories.*
**os.fsencode(filename)**

Encode path-like filename to the filesystem encoding with 'surrogateescape' error handler, or 'strict' on Windows; return bytes unchanged.

fsdecode() is the reverse function.

*New in version 3.2.*

*Changed in version 3.6:* Support added to accept objects implementing the os.PathLike interface.

**os.fsdecode(filename)**

Decode the path-like filename from the filesystem encoding with 'surrogateescape' error handler, or 'strict' on Windows; return str unchanged.

fsencode() is the reverse function.

*New in version 3.2.*

*Changed in version 3.6:* Support added to accept objects implementing the os.PathLike interface.

**os.fspath(path)**

Return the file system representation of the path.

If str or bytes is passed in, it is returned unchanged. Otherwise __fspath__() is called and its value is returned as long as it is a str or bytes object. In all other cases, TypeError is raised.

*New in version 3.6.*

**class os.PathLike**
An **abstract base class** for objects representing a file system path, e.g. `pathlib.PurePath`.

*New in version 3.6.*

**abstractmethod** `__fspath__()`

Return the file system path representation of the object.

The method should only return a `str` or `bytes` object, with the preference being for `str`.

```python
os.getenv(key, default=None)
```

Return the value of the environment variable `key` if it exists, or `default` if it doesn’t. `key`, `default` and the result are `str`.

On Unix, keys and values are decoded with `sys.getfilesystemencoding()` and `'surrogateescape'` error handler. Use `os.getenvb()` if you would like to use a different encoding.

**Availability**: most flavors of Unix, Windows.

```python
os.getenvb(key, default=None)
```

Return the value of the environment variable `key` if it exists, or `default` if it doesn’t. `key`, `default` and the result are `bytes`.

`getenvb()` is only available if `supports_bytes_environ` is True.

**Availability**: most flavors of Unix.

*New in version 3.2.*

```python
os.get_exec_path(env=None)
```
Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. env, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when env is None, environ is used.

*New in version 3.2.*

**os.getegid()**
Return the effective group id of the current process. This corresponds to the “set id” bit on the file being executed in the current process.

*Availability:* Unix.

**os.geteuid()**
Return the current process’s effective user id.

*Availability:* Unix.

**os.getgid()**
Return the real group id of the current process.

*Availability:* Unix.

**os.getgrouplist(user, group)**
Return list of group ids that user belongs to. If group is not in the list, it is included; typically, group is specified as the group ID field from the password record for user.

*Availability:* Unix.

*New in version 3.3.*

**os.getgroups()**
Return list of supplemental group ids associated with the current process.

**Availability:** Unix.

**Note:** On Mac OS X, `getgroups()` behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of 10.5 or earlier, `getgroups()` returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to `setgroups()` if suitably privileged. If built with a deployment target greater than 10.5, `getgroups()` returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to `setgroups()`, and its length is not limited to 16. The deployment target value, MACOSX_DEPLOYMENT_TARGET, can be obtained with `sysconfig.get_config_var()`.

`os.getlogin()`

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use `getpass.getuser()` since the latter checks the environment variables LOGNAME or USERNAME to find out who the user is, and falls back to `pwd.getpwuid(os.getuid())[0]` to get the login name of the current real user id.

**Availability:** Unix, Windows.

`os.getpgid(pid)`

Return the process group id of the process with process id `pid`. If
*pid* is 0, the process group id of the current process is returned.

**Availability:** Unix.

**os. getpgrp()**
Return the id of the current process group.

**Availability:** Unix.

**os. getpid()**
Return the current process id.

**os. getppid()**
Return the parent’s process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

**Availability:** Unix, Windows.

*Changed in version 3.2:* Added support for Windows.

**os. getpriority**(which, who)
Get program scheduling priority. The value *which* is one of PRI0_PROCESS, PRI0_PGRP, or PRI0_USER, and *who* is interpreted relative to *which* (a process identifier for PRI0_PROCESS, process group identifier for PRI0_PGRP, and a user ID for PRI0_USER). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

**Availability:** Unix.

*New in version 3.3.*
**os.
PRIO_PROCESS**

**os.
PRIO_PGRP**

**os.
PRIO_USER**

Parameters for the `getpriority()` and `setpriority()` functions.

*Availability:* Unix.

*New in version 3.3.*

**os.
getresuid()**

Return a tuple `(ruid, euid, suid)` denoting the current process’s real, effective, and saved user ids.

*Availability:* Unix.

*New in version 3.2.*

**os.
getresgid()**

Return a tuple `(rgid, egid, sgid)` denoting the current process’s real, effective, and saved group ids.

*Availability:* Unix.

*New in version 3.2.*

**os.
getuid()**

Return the current process’s real user id.

*Availability:* Unix.

**os.
initgroups(username, gid)**

Call the system `initgroups()` to initialize the group access list with all of the groups of which the specified username is a member, plus
the specified group id.

**Availability:** Unix.

*New in version 3.2.*

**os.putenv**(key, value)

Set the environment variable named `key` to the string `value`. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

**Availability:** most flavors of Unix, Windows.

**Note:** On some platforms, including FreeBSD and Mac OS X, setting environ may cause memory leaks. Refer to the system documentation for `putenv`.

When `putenv()` is supported, assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`; however, calls to `putenv()` don't update `os.environ`, so it is actually preferable to assign to items of `os.environ`.

**os.setegid**(egid)

Set the current process's effective group id.

**Availability:** Unix.

**os.seteuid**(euid)

Set the current process's effective user id.

**Availability:** Unix.

**os.setgid**(gid)
Set the current process’ group id.

**Availability:** Unix.

```python
os.setgroups(groups)
```

Set the list of supplemental group ids associated with the current process to `groups`. `groups` must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

**Availability:** Unix.

**Note:** On Mac OS X, the length of `groups` may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling `setgroups()`.

```python
os.setpgrp()
```

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

**Availability:** Unix.

```python
os.setpgid(pid, pgrp)
```

Call the system call `setpgid()` to set the process group id of the process with id `pid` to the process group with id `pgrp`. See the Unix manual for the semantics.

**Availability:** Unix.

```python
os.setpriority(which, who, priority)
```

Set program scheduling priority. The value `which` is one of
PRI0_PROCESS, PRI0_PGRP, or PRI0_USER, and who is interpreted relative to which (a process identifier for PRI0_PROCESS, process group identifier for PRI0_PGRP, and a user ID for PRI0_USER). A zero value for who denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. priority is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

Availability: Unix.

New in version 3.3.

**os.setregid**(*rgid, egid*)
Set the current process’s real and effective group ids.

Availability: Unix.

**os.setresgid**(*rgid, egid, sgid*)
Set the current process’s real, effective, and saved group ids.

Availability: Unix.

New in version 3.2.

**os.setresuid**(*ruid, euid, suid*)
Set the current process’s real, effective, and saved user ids.

Availability: Unix.

New in version 3.2.

**os.setreuid**(*ruid, euid*)
Set the current process’s real and effective user ids.
Availability: Unix.

os.**getsid**(pid)
Call the system call getsid(). See the Unix manual for the semantics.

Availability: Unix.

os.**setsid**()
Call the system call setsid(). See the Unix manual for the semantics.

Availability: Unix.

os.**setuid**(uid)
Set the current process’s user id.

Availability: Unix.

os.**strerror**(code)
Return the error message corresponding to the error code in code. On platforms where strerror() returns NULL when given an unknown error number, ValueError is raised.

os.**supports_bytes_environ**
True if the native OS type of the environment is bytes (eg. False on Windows).

*New in version 3.2.*

os.**umask**(mask)
Set the current numeric umask and return the previous umask.

os.**uname**()
Returns information identifying the current operating system. The return value is an object with five attributes:

- sysname - operating system name
- nodename - name of machine on network (implementation-defined)
- release - operating system release
- version - operating system version
- machine - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing sysname, nodename, release, version, and machine in that order.

Some systems truncate nodename to 8 characters or to the leading component; a better way to get the hostname is `socket.gethostname()` or even `socket.gethostbyaddr(socket.gethostname())`.

**Availability:** recent flavors of Unix.

*Changed in version 3.3:* Return type changed from a tuple to a tuple-like object with named attributes.

**os.unsetenv(key)**

Unset (delete) the environment variable named `key`. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

When `unsetenv()` is supported, deletion of items in `os.environ` is automatically translated into a corresponding call to `unsetenv()`; however, calls to `unsetenv()` don't update `os.environ`, so it is actually preferable to delete items of
os.environ.

**Availability**: most flavors of Unix, Windows.
File Object Creation

This function creates new file objects. (See also open() for opening file descriptors.)

os. fdopen(fd, *args, **kwargs)

Return an open file object connected to the file descriptor fd. This is an alias of the open() built-in function and accepts the same arguments. The only difference is that the first argument of fdopen() must always be an integer.
File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name “file descriptor” is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

**os.close(fd)**

Close file descriptor `fd`.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

**os.closerange(fd_low, fd_high)**

Close all file descriptors from `fd_low` (inclusive) to `fd_high` (exclusive), ignoring errors. Equivalent to (but much faster than):

```python
for fd in range(fd_low, fd_high):
    try:
```
```python
os.close(fd)
except OSError:
    pass
```

**os.device_encoding(fd)**

Return a string describing the encoding of the device associated with `fd` if it is connected to a terminal; else return `None`.

**os.dup(fd)**

Return a duplicate of file descriptor `fd`. The new file descriptor is **non-inheritable**.

On Windows, when duplicating a standard stream (0: stdin, 1: stdout, 2: stderr), the new file descriptor is **inheritable**.

*Changed in version 3.4:* The new file descriptor is now non-inheritable.

**os.dup2(fd, fd2, inheritable=True)**

Duplicate file descriptor `fd` to `fd2`, closing the latter first if necessary. Return `fd2`. The new file descriptor is **inheritable** by default or non-inheritable if `inheritable` is `False`.

*Changed in version 3.4:* Add the optional `inheritable` parameter.

*Changed in version 3.7:* Return `fd2` on success. Previously, `None` was always returned.

**os.fchmod(fd, mode)**

Change the mode of the file given by `fd` to the numeric `mode`. See the docs for `chmod()` for possible values of `mode`. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

**Availability:** Unix.
os.\texttt{fchown}(fd, uid, gid)

Change the owner and group id of the file given by \texttt{fd} to the numeric \texttt{uid} and \texttt{gid}. To leave one of the ids unchanged, set it to \texttt{-1}. See \texttt{chown()}. As of Python 3.3, this is equivalent to \texttt{os.chown(fd, uid, gid)}.

\textbf{Availability:} Unix.

os.\texttt{fdatasync}(fd)

Force write of file with filedescriptor \texttt{fd} to disk. Does not force update of metadata.

\textbf{Availability:} Unix.

\begin{quote}
\textbf{Note:} This function is not available on MacOS.
\end{quote}

os.\texttt{fpathconf}(fd, name)

Return system configuration information relevant to an open file. \texttt{name} specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the \texttt{pathconf_names} dictionary. For configuration variables not included in that mapping, passing an integer for \texttt{name} is also accepted.

If \texttt{name} is a string and is not known, \texttt{ValueError} is raised. If a specific value for \texttt{name} is not supported by the host system, even if it is included in \texttt{pathconf_names}, an \texttt{OSError} is raised with \texttt{errno.EINVAL} for the error number.
As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

**Availability:** Unix.

---

**os.fstat(fd)**

Get the status of the file descriptor `fd`. Return a `stat_result` object.

As of Python 3.3, this is equivalent to `os.stat(fd)`.

**See also:** The `stat()` function.

---

**os.fstatvfs(fd)**

Return information about the filesystem containing the file associated with file descriptor `fd`, like `statvfs()`. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

**Availability:** Unix.

---

**os.fsync(fd)**

Force write of file with filedescriptor `fd` to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `commit()` function.

If you're starting with a buffered Python file object `f`, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with `f` are written to disk.

**Availability:** Unix, Windows.

---

**os.ftruncate(fd, length)**

Truncate the file corresponding to file descriptor `fd`, so that it is at most `length` bytes in size. As of Python 3.3, this is equivalent to
os.truncate(fd, length).

Availability: Unix, Windows.

Changed in version 3.5: Added support for Windows

os.get_blocking(fd)
Get the blocking mode of the file descriptor: False if the O_NONBLOCK flag is set, True if the flag is cleared.

See also set_blocking() and socket.socket.setblocking().

Availability: Unix.

New in version 3.5.

os.isatty(fd)
Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.

os.lockf(fd, cmd, len)
Apply, test or remove a POSIX lock on an open file descriptor. fd is an open file descriptor. cmd specifies the command to use - one of F_LOCK, F_TLOCK, F_ULOCK or F_TEST. len specifies the section of the file to lock.

Availability: Unix.

New in version 3.3.

os.F_LOCK
os.F_TLOCK
os.F_ULOCK
os. `F_TEST`
Flags that specify what action `lockf()` will take.

**Availability**: Unix.

*New in version 3.3.*

os. `lseek(fd, pos, how)`
Set the current position of file descriptor `fd` to position `pos`, modified by `how`: `SEEK_SET` or 0 to set the position relative to the beginning of the file; `SEEK_CUR` or 1 to set it relative to the current position; `SEEK_END` or 2 to set it relative to the end of the file. Return the new cursor position in bytes, starting from the beginning.

os. `SEEK_SET`
os. `SEEK_CUR`
os. `SEEK_END`
Parameters to the `lseek()` function. Their values are 0, 1, and 2, respectively.

*New in version 3.3*: Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`.

os. `open(path, flags, mode=0o777, *, dir_fd=None)`
Open the file `path` and set various flags according to `flags` and possibly its mode according to `mode`. When computing `mode`, the current umask value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is non-inheritable.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like `O_RDONLY` and `O_WRONLY`) are defined in the `os` module. In particular, on Windows adding `O_BINARY` is needed to open files in binary mode.
This function can support paths relative to directory descriptors with the *dir_fd* parameter.

*Changed in version 3.4:* The new file descriptor is now non-inheritable.

**Note:** This function is intended for low-level I/O. For normal usage, use the built-in function `open()`, which returns a file object with `read()` and `write()` methods (and many more). To wrap a file descriptor in a file object, use `fdopen()`.

*New in version 3.3:* The *dir_fd* argument.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

*Changed in version 3.6:* Accepts a path-like object.

The following constants are options for the *flags* parameter to the `open()` function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the `open(2)` manual page on Unix or the MSDN on Windows.

```python
os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
```
The above constants are available on Unix and Windows.

```
   os.0_DSYNC
   os.0_RSYNC
   os.0_SYNC
   os.0_NDELAY
   os.0_NONBLOCK
   os.0_NOCTTY
   os.0_CLOEXEC
```

The above constants are only available on Unix.

*Changed in version 3.3: Add 0_CLOEXEC constant.*

```
   os.0_BINARY
   os.0_NOINHERIT
   os.0_SHORT_LIVED
   os.0_TEMPORARY
   os.0_RANDOM
   os.0_SEQUENCEAL
   os.0_TEXT
```

The above constants are only available on Windows.

```
   os.0_ASYNC
   os.0_DIRECT
   os.0_DIRECTORY
   os.0_NOFOLLOW
   os.0_NOATIME
   os.0_PATH
   os.0_TMPFILE
   os.0_SHLOCK
   os.0_EXLOCK
```

The above constants are extensions and not present if they are not defined by the C library.
Changed in version 3.4: Add `O_PATH` on systems that support it. Add `O_TMPFILE`, only available on Linux Kernel 3.11 or newer.

**os.** `openpty()`  
Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. The new file descriptors are non-inheritable. For a (slightly) more portable approach, use the pty module.

**Availability:** some flavors of Unix.

Changed in version 3.4: The new file descriptors are now non-inheritable.

**os.** `pipe()`  
Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively. The new file descriptor is non-inheritable.

**Availability:** Unix, Windows.

Changed in version 3.4: The new file descriptors are now non-inheritable.

**os.** `pipe2(flags)`  
Create a pipe with flags set atomically. flags can be constructed by ORing together one or more of these values: `O_NONBLOCK`, `O_CLOEXEC`. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.

**Availability:** some flavors of Unix.

New in version 3.3.
os. `posix_fallocate(fd, offset, len)`

Ensures that enough disk space is allocated for the file specified by `fd` starting from `offset` and continuing for `len` bytes.

**Availability**: Unix.

*New in version 3.3.*

os. `posix_fadvise(fd, offset, len, advice)`

Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by `fd` starting at `offset` and continuing for `len` bytes. `advice` is one of `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED` or `POSIX_FADV_DONTNEED`.

**Availability**: Unix.

*New in version 3.3.*

os. `POSIX_FADV_NORMAL`

os. `POSIX_FADV_SEQUENTIAL`

os. `POSIX_FADV_RANDOM`

os. `POSIX_FADV_NOREUSE`

os. `POSIX_FADV_WILLNEED`

os. `POSIX_FADV_DONTNEED`

Flags that can be used in `advice` in `posix_fadvise()` that specify the access pattern that is likely to be used.

**Availability**: Unix.

*New in version 3.3.*
os. **pread**(fd, n, offset)
Read at most n bytes from file descriptor fd at a position of offset, leaving the file offset unchanged.

Return a bytestring containing the bytes read. If the end of the file referred to by fd has been reached, an empty bytes object is returned.

**Availability:** Unix.

**New in version 3.3.**

os. **preadv**(fd, buffers, offset, flags=0)
Read from a file descriptor fd at a position of offset into mutable bytes-like objects buffers, leaving the file offset unchanged. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

The flags argument contains a bitwise OR of zero or more of the following flags:

- RWF_HIPRI
- RWF_NOWAIT

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (sysconf() value 'SC_IOV_MAX') on the number of buffers that can be used.

Combine the functionality of os.readv() and os.pread().

**Availability:** Linux 2.6.30 and newer, FreeBSD 6.0 and newer, OpenBSD 2.7 and newer. Using flags requires Linux 4.6 or newer.
New in version 3.7.

**os. RWF_NOWAIT**

Do not wait for data which is not immediately available. If this flag is specified, the system call will return instantly if it would have to read data from the backing storage or wait for a lock.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return \(-1\) and set errno to \(\text{errno.EAGAIN}\).

**Availability:** Linux 4.14 and newer.

New in version 3.7.

**os. RWF_HIPRI**

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources.

Currently, on Linux, this feature is usable only on a file descriptor opened using the \(\text{O_DIRECT}\) flag.

**Availability:** Linux 4.6 and newer.

New in version 3.7.

**os. pwrite\((fd, str, offset)\)**

Write the bytestring in \(str\) to file descriptor \(fd\) at position of \(offset\), leaving the file offset unchanged.

Return the number of bytes actually written.

**Availability:** Unix.
New in version 3.3.

```python
os.pwritev(fd, buffers, offset, flags=0)
```
Write the `buffers` contents to file descriptor `fd` at a offset `offset`, leaving the file offset unchanged. `buffers` must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

The flags argument contains a bitwise OR of zero or more of the following flags:

- `RWF_DSYNC`
- `RWF_SYNC`

Return the total number of bytes actually written.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of `os.writev()` and `os.pwrite()`.  

**Availability:** Linux 2.6.30 and newer, FreeBSD 6.0 and newer, OpenBSD 2.7 and newer. Using flags requires Linux 4.7 or newer.

New in version 3.7.

```python
os.RWF_DSYNC
```
Provide a per-write equivalent of the `O_DSYNC open(2)` flag. This flag effect applies only to the data range written by the system call.

**Availability:** Linux 4.7 and newer.

New in version 3.7.
**os.RWF_SYNC**
Provide a per-write equivalent of the `O_SYNC` `open(2)` flag. This flag effect applies only to the data range written by the system call.

**Availability:** Linux 4.7 and newer.

*New in version 3.7.*

**os.read(fd, n)**
Read at most `n` bytes from file descriptor `fd`.

Return a bytestring containing the bytes read. If the end of the file referred to by `fd` has been reached, an empty bytes object is returned.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To read a “file object” returned by the built-in function `open()`, or by `popen()` or `fdopen()`, or `sys.stdin`, use its `read()` or `readline()` methods.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

**os.sendfile(out, in, offset, count)**

**os.sendfile(out, in, offset, count, [headers], [trailers], flags=0)**
Copy `count` bytes from file descriptor `in` to file descriptor `out` starting at `offset`. Return the number of bytes sent. When EOF is reached return 0.

The first function notation is supported by all platforms that define
**sendfile()**

On Linux, if `offset` is given as `None`, the bytes are read from the current position of `in` and the position of `in` is updated.

The second case may be used on Mac OS X and FreeBSD where `headers` and `trailers` are arbitrary sequences of buffers that are written before and after the data from `in` is written. It returns the same as the first case.

On Mac OS X and FreeBSD, a value of 0 for `count` specifies to send until the end of `in` is reached.

All platforms support sockets as `out` file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Cross-platform applications should not use `headers`, `trailers` and `flags` arguments.

**Availability:** Unix.

**Note:** For a higher-level wrapper of `sendfile()`, see `socket.socket.sendfile()`.

*New in version 3.3.*

**os.set_blocking(fd, blocking)**

Set the blocking mode of the specified file descriptor. Set the `O_NONBLOCK` flag if blocking is `False`, clear the flag otherwise.

See also `get_blocking()` and `socket.socket.setblocking()`.

**Availability:** Unix.
New in version 3.5.

**os.SF_NODISKIO**

**os.SF_MNOWAIT**

**os.SF_SYNC**

Parameters to the `sendfile()` function, if the implementation supports them.

**Availability:** Unix.

New in version 3.3.

**os.readv(fd, buffers)**

Read from a file descriptor `fd` into a number of mutable bytes-like objects `buffers`. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

**Availability:** Unix.

New in version 3.3.

**os.tcgetpgrp(fd)**

Return the process group associated with the terminal given by `fd` (an open file descriptor as returned by `os.open()`).

**Availability:** Unix.
os. `tcsetpgrp(fd, pg)`
Set the process group associated with the terminal given by `fd` (an open file descriptor as returned by `os.open()`) to `pg`.

**Availability:** Unix.

os. `ttyname(fd)`
Return a string which specifies the terminal device associated with file descriptor `fd`. If `fd` is not associated with a terminal device, an exception is raised.

**Availability:** Unix.

os. `write(fd, str)`
Write the bytestring in `str` to file descriptor `fd`.

Return the number of bytes actually written.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To write a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

os. `writev(fd, buffers)`
Write the contents of `buffers` to file descriptor `fd`. `buffers` must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding
to the second, and so on.

Returns the total number of bytes actually written.

The operating system may set a limit (sysconf() value 'SC_IOV_MAX') on the number of buffers that can be used.

**Availability:** Unix.

*New in version 3.3.*

**Querying the size of a terminal**

*New in version 3.3.*

**os.get_terminal_size**(fd=STDOUT_FILENO)

Return the size of the terminal window as (columns, lines), tuple of type `terminal_size`.

The optional argument `fd` (default `STDOUT_FILENO`, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an `OSError` is raised.

`shutil.get_terminal_size()` is the high-level function which should normally be used, `os.get_terminal_size` is the low-level implementation.

**Availability:** Unix, Windows.

**class os.terminal_size**

A subclass of tuple, holding (columns, lines) of the terminal
window size.

columns
   Width of the terminal window in characters.

lines
   Height of the terminal window in characters.

Inheritance of File Descriptors

New in version 3.4.

A file descriptor has an “inheritable” flag which indicates if the file descriptor can be inherited by child processes. Since Python 3.4, file descriptors created by Python are non-inheritable by default.

On UNIX, non-inheritable file descriptors are closed in child processes at the execution of a new program, other file descriptors are inherited.

On Windows, non-inheritable handles and file descriptors are closed in child processes, except for standard streams (file descriptors 0, 1 and 2: stdin, stdout and stderr), which are always inherited. Using spawn* functions, all inheritable handles and all inheritable file descriptors are inherited. Using the subprocess module, all file descriptors except standard streams are closed, and inheritable handles are only inherited if the close_fds parameter is False.

os.get_inheritable(fd)
   Get the “inheritable” flag of the specified file descriptor (a boolean).

os.set_inheritable(fd, inheritable)
   Set the “inheritable” flag of the specified file descriptor.
**os.get_handle_inheritable**(handle)

Get the “inheritable” flag of the specified handle (a boolean).

**Availability**: Windows.

**os.set_handle_inheritable**(handle, inheritable)

Set the “inheritable” flag of the specified handle.

**Availability**: Windows.
Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** For some functions, the `path` argument can be not only a string giving a path name, but also a file descriptor. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the \_\_at or \_f\_at version of the function.)

You can check whether or not `path` can be specified as a file descriptor on your platform using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

If the function also supports `dir_fd` or `follow_symlinks` arguments, it is an error to specify one of those when supplying `path` as a file descriptor.

- **paths relative to directory descriptors:** If `dir_fd` is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, `dir_fd` is ignored. (For POSIX systems, Python will call the \_\_at or \_f\_at version of the function.)

You can check whether or not `dir_fd` is supported on your platform using `os.supports_dir_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

- **not following symlinks:** If `follow_symlinks` is `False`, and the last element of the path to operate on is a symbolic link, the function
will operate on the symbolic link itself instead of the file the link points to. (For POSIX systems, Python will call the ./ version of the function.)

You can check whether or not follow_symlinks is supported on your platform using os.supports_follow_symlinks. If it is unavailable, using it will raise a NotImplementedError.

```python
os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)
```

Use the real uid/gid to test for access to `path`. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to `path`. `mode` should be F_OK to test the existence of `path`, or it can be the inclusive OR of one or more of R_OK, W_OK, and X_OK to test permissions. Return True if access is allowed, False if not. See the Unix man page access(2) for more information.

This function can support specifying paths relative to directory descriptors and not following symlinks.

If `effective_ids` is True, access() will perform its access checks using the effective uid/gid instead of the real uid/gid. `effective_ids` may not be supported on your platform; you can check whether or not it is available using os.supports_effective_ids. If it is unavailable, using it will raise a NotImplementedError.

**Note:** Using access() to check if a user is authorized to e.g. open a file before actually doing so using open() creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it.
It's preferable to use EAFP techniques. For example:

```python
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()

return "some default data"
```

is better written as:

```python
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

**Note:** I/O operations may fail even when `access()` indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

*Changed in version 3.3:* Added the `dir_fd`, `effective_ids`, and `follow_symlinks` parameters.

*Changed in version 3.6:* Accepts a path-like object.

- `os.F_OK`
- `os.R_OK`
- `os.W_OK`
- `os.X_OK`

Values to pass as the `mode` parameter of `access()` to test the existence, readability, writability and executability of `path`, respectively.
os.chdir(path)
Change the current working directory to path.

This function can support specifying a file descriptor. The descriptor must refer to an opened directory, not an open file.

New in version 3.3: Added support for specifying path as a file descriptor on some platforms.

Changed in version 3.6: Accepts a path-like object.

os.chflags(path, flags, *, follow_symlinks=True)
Set the flags of path to the numeric flags. flags may take a combination (bitwise OR) of the following values (as defined in the stat module):

- stat.UF_NODUMP
- stat.UF_IMMUTABLE
- stat.UF_APPEND
- stat.UF_OPAQUE
- stat.UF_NOUNLINK
- stat.UF_COMPRESSED
- stat.UF_HIDDEN
- stat.SF_ARCHIVED
- stat.SF_IMMUTABLE
- stat.SF_APPEND
- stat.SF_NOUNLINK
- stat.SF_SNAPSHOT

This function can support not following symlinks.

Availability: Unix.

New in version 3.3: The follow_symlinks argument.
Changed in version 3.6: Accepts a path-like object.

```python
os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True)
```

Change the mode of `path` to the numeric `mode`. `mode` may take one of the following values (as defined in the `stat` module) or bitwise ORed combinations of them:

- `stat.S_ISUID`
- `stat.S_ISGID`
- `stat.S_ENFMT`
- `stat.S_ISVTX`
- `stat.S_IREAD`
- `stat.S_IWRITE`
- `stat.S_IEXEC`
- `stat.S_IRWXU`
- `stat.S_IRUSR`
- `stat.S_IWUSR`
- `stat.S_IXUSR`
- `stat.S_IRWXG`
- `stat.S_IRGRP`
- `stat.S_IWGRP`
- `stat.S_IXGRP`
- `stat.S_IRWXO`
- `stat.S_IROTH`
- `stat.S_IWOTH`
- `stat.S_IXOTH`

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

**Note:** Although Windows supports `chmod()`, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and
New in version 3.3: Added support for specifying `path` as an open file descriptor, and the `dir_fd` and `follow_symlinks` arguments.

Changed in version 3.6: Accepts a **path-like object**.

```python
cos.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)
```
Change the owner and group id of `path` to the numeric `uid` and `gid`. To leave one of the ids unchanged, set it to -1.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

See `shutil.chown()` for a higher-level function that accepts names in addition to numeric ids.

**Availability:** Unix.

New in version 3.3: Added support for specifying an open file descriptor for `path`, and the `dir_fd` and `follow_symlinks` arguments.

Changed in version 3.6: Supports a **path-like object**.

```python
cos.chroot(path)
```
Change the root directory of the current process to `path`.

**Availability:** Unix.

Changed in version 3.6: Accepts a **path-like object**.

```python
cos.fchdir(fd)
```
Change the current working directory to the directory represented by the file descriptor `fd`. The descriptor must refer to an opened
directory, not an open file. As of Python 3.3, this is equivalent to
\texttt{os.chdir(fd)}.

\textit{Availability}: Unix.

\texttt{os.getcwd()}

Return a string representing the current working directory.

\texttt{os.getcwdb()}

Return a bytestring representing the current working directory.

\texttt{os.lchflags(path, flags)}

Set the flags of \texttt{path} to the numeric \texttt{flags}, like \texttt{chflags()}, but do not follow symbolic links. As of Python 3.3, this is equivalent to \texttt{os.chflags(path, flags, follow_symlinks=False)}.

\textit{Availability}: Unix.

\textit{Changed in version 3.6}: Accepts a \texttt{path-like object}.

\texttt{os.lchmod(path, mode)}

Change the mode of \texttt{path} to the numeric \texttt{mode}. If \texttt{path} is a symlink, this affects the symlink rather than the target. See the docs for \texttt{chmod()} for possible values of \texttt{mode}. As of Python 3.3, this is equivalent to \texttt{os.chmod(path, mode, follow_symlinks=False)}.

\textit{Availability}: Unix.

\textit{Changed in version 3.6}: Accepts a \texttt{path-like object}.

\texttt{os.lchown(path, uid, gid)}

Change the owner and group id of \texttt{path} to the numeric \texttt{uid} and \texttt{gid}. This function will not follow symbolic links. As of Python 3.3, this is
equivalent to `os.chown(path, uid, gid, follow_symlinks=False)`.

**Availability:** Unix.

*Changed in version 3.6:* Accepts a path-like object.

`os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)`

Create a hard link pointing to `src` named `dst`.

This function can support specifying `src_dir_fd` and/or `dst_dir_fd` to supply paths relative to directory descriptors, and not following symlinks.

**Availability:** Unix, Windows.

*Changed in version 3.2:* Added Windows support.

*New in version 3.3:* Added the `src_dir_fd`, `dst_dir_fd`, and `follow_symlinks` arguments.

*Changed in version 3.6:* Accepts a path-like object for `src` and `dst`.

`os.listdir(path='.')`

Return a list containing the names of the entries in the directory given by `path`. The list is in arbitrary order, and does not include the special entries `''` and `..'` even if they are present in the directory.

`path` may be a path-like object. If `path` is of type bytes (directly or indirectly through the PathLike interface), the filenames returned will also be of type bytes; in all other circumstances, they will be of type str.
This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

**Note:** To encode str filenames to bytes, use `fsencode()`.

**See also:** The `scandir()` function returns directory entries along with file attribute information, giving better performance for many common use cases.

*Changed in version 3.2:* The `path` parameter became optional.

*New in version 3.3:* Added support for specifying an open file descriptor for `path`.

*Changed in version 3.6:* Accepts a path-like object.

```
import os

os.lstat(path, *, dir_fd=None)
```

Perform the equivalent of an `lstat()` system call on the given path. Similar to `stat()`, but does not follow symbolic links. Return a `stat_result` object.

On platforms that do not support symbolic links, this is an alias for `stat()`.

As of Python 3.3, this is equivalent to `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

This function can also support paths relative to directory descriptors.

**See also:** The `stat()` function.

*Changed in version 3.2:* Added support for Windows 6.0 (Vista)
symbolic links.

*Changed in version 3.3:* Added the `dir_fd` parameter.

*Changed in version 3.6:* Accepts a path-like object for `src` and `dst`.

```python
os.mkdir(path, mode=0o777, *, dir_fd=None)
```

Create a directory named `path` with numeric mode `mode`.

If the directory already exists, `FileExistsError` is raised.

On some systems, `mode` is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the `mode`) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call `chmod()` explicitly to set them.

This function can also support paths relative to directory descriptors.

It is also possible to create temporary directories; see the `tempfile` module’s `tempfile.mkdtemp()` function.

*New in version 3.3:* The `dir_fd` argument.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.makedirs(name, mode=0o777, exist_ok=False)
```

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The `mode` parameter is passed to `mkdir()` for creating the leaf directory; see the `mkdir()` description for how it is interpreted. To set the file permission bits of any newly-created parent directories you
can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If `exist_ok` is `False` (the default), an `OSError` is raised if the target directory already exists.

**Note:** `makedirs()` will become confused if the path elements to create include `pardir` (eg. “..” on UNIX systems).

This function handles UNC paths correctly.

*New in version 3.2:* The `exist_ok` parameter.

*Changed in version 3.4.1:* Before Python 3.4.1, if `exist_ok` was `True` and the directory existed, `makedirs()` would still raise an error if `mode` did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See [bpo-21082](https://bugs.python.org/issue21082).

*Changed in version 3.6:* Accepts a *path-like object*.

*Changed in version 3.7:* The `mode` argument no longer affects the file permission bits of newly-created intermediate-level directories.

```python
os.mkfifo(path, mode=0o666, *, dir_fd=None)
```

Create a FIFO (a named pipe) named `path` with numeric mode `mode`. The current umask value is first masked out from the mode.

This function can also support *paths relative to directory descriptors*.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between “client” and “server” type
processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn't open the FIFO — it just creates the rendezvous point.

**Availability:** Unix.

*New in version 3.3:* The `dir_fd` argument.

*Changed in version 3.6:* Accepts a **path-like object**.

```python
os.mknod(path, mode=0o600, device=0, *, dir_fd=None)
```

Create a filesystem node (file, device special file or named pipe) named `path`. `mode` specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in `stat`). For `stat.S_IFCHR` and `stat.S_IFBLK`, `device` defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

This function can also support **paths relative to directory descriptors**.

**Availability:** Unix.

*New in version 3.3:* The `dir_fd` argument.

*Changed in version 3.6:* Accepts a **path-like object**.

```python
os.major(device)
```

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

```python
os.minor(device)
```
Extract the device minor number from a raw device number (usually the st_dev or st_rdev field from stat).

```python
os.makedev(major, minor)
```

Compose a raw device number from the major and minor device numbers.

```python
os.pathconf(path, name)
```

Return system configuration information relevant to a named file. `name` specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for `name` is also accepted.

If `name` is a string and is not known, `ValueError` is raised. If a specific value for `name` is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

This function can support specifying a file descriptor.

**Availability:** Unix.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.pathconf_names
```

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set
of names known to the system.

**Availability:** Unix.

```python
os.readlink(path, *, dir_fd=None)
```

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the `path` is a string object (directly or indirectly through a `PathLike` interface), the result will also be a string object, and the call may raise a `UnicodeDecodeError`. If the `path` is a bytes object (direct or indirectly), the result will be a bytes object.

This function can also support paths relative to directory descriptors.

**Availability:** Unix, Windows.

*Changed in version 3.2:* Added support for Windows 6.0 (Vista) symbolic links.

*New in version 3.3:* The `dir_fd` argument.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.remove(path, *, dir_fd=None)
```

Remove (delete) the file `path`. If `path` is a directory, `OSError` is raised. Use `rmdir()` to remove directories.

This function can support paths relative to directory descriptors.

On Windows, attempting to remove a file that is in use causes an
exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is semantically identical to `unlink()`.

*New in version 3.3:* The `dir_fd` argument.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.removedirs(name)
```

Remove directories recursively. Works like `rmdir()` except that, if the leaf directory is successfully removed, `removedirs()` tries to successively remove every parent directory mentioned in `path` until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory 'foo/bar/baz', and then remove 'foo/bar' and 'foo' if they are empty. Raises `OSError` if the leaf directory could not be successfully removed.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
```

Rename the file or directory `src` to `dst`. If `dst` is a directory, `OSError` will be raised. On Unix, if `dst` exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if `src` and `dst` are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if `dst` already exists, `OSError` will be raised even if it is a file.

This function can support specifying `src_dir_fd` and/or `dst_dir_fd` to
supply paths relative to directory descriptors.

If you want cross-platform overwriting of the destination, use replace().

*New in version 3.3:* The src_dir_fd and dst_dir_fd arguments.

*Changed in version 3.6:* Accepts a path-like object for src and dst.

```
import os

os.renames(old, new)
```

Recursive directory or file renaming function. Works like rename(), except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removedirs().

**Note:** This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

*Changed in version 3.6:* Accepts a path-like object for old and new.

```
import os

os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
```

Rename the file or directory src to dst. If dst is a directory, OSError will be raised. If dst exists and is a file, it will be replaced silently if the user has permission. The operation may fail if src and dst are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying src_dir_fd and/or dst_dir_fd to supply paths relative to directory descriptors.

*New in version 3.3.*
Changed in version 3.6: Accepts a path-like object for src and dst.

`os.rmdir(path, *, dir_fd=None)`

Remove (delete) the directory path. Only works when the directory is empty, otherwise, `OSError` is raised. In order to remove whole directory trees, `shutil.rmtree()` can be used.

This function can support paths relative to directory descriptors.

New in version 3.3: The dir_fd parameter.

Changed in version 3.6: Accepts a path-like object.

`os.scandir(path='.' )`

Return an iterator of `os.DirEntry` objects corresponding to the entries in the directory given by path. The entries are yielded in arbitrary order, and the special entries '.' and '..' are not included.

Using `scandir()` instead of `listdir()` can significantly increase the performance of code that also needs file type or file attribute information, because `os.DirEntry` objects expose this information if the operating system provides it when scanning a directory. All `os.DirEntry` methods may perform a system call, but `is_dir()` and `is_file()` usually only require a system call for symbolic links; `os.DirEntry.stat()` always requires a system call on Unix but only requires one for symbolic links on Windows.

Path may be a path-like object. If path is of type bytes (directly or indirectly through the PathLike interface), the type of the name and path attributes of each `os.DirEntry` will be bytes; in all other circumstances, they will be of type str.
This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

The `scandir()` iterator supports the context manager protocol and has the following method:

```
scandir.close()
```

Close the iterator and free acquired resources.

This is called automatically when the iterator is exhausted or garbage collected, or when an error happens during iterating. However it is advisable to call it explicitly or use the `with` statement.

*New in version 3.6.*

The following example shows a simple use of `scandir()` to display all the files (excluding directories) in the given `path` that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```python
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

*Note:* On Unix-based systems, `scandir()` uses the system's `opendir()` and `readdir()` functions. On Windows, it uses the Win32 `FindFirstFileW` and `FindNextFileW` functions.

*New in version 3.5.*
New in version 3.6: Added support for the context manager protocol and the `close()` method. If a `scandir()` iterator is neither exhausted nor explicitly closed a `ResourceWarning` will be emitted in its destructor.

The function accepts a path-like object.

Changed in version 3.7: Added support for file descriptors on Unix.

```python
class os.DirEntry

Object yielded by `scandir()` to expose the file path and other file attributes of a directory entry.

`scandir()` will provide as much of this information as possible without making additional system calls. When a `stat()` or `lstat()` system call is made, the `os.DirEntry` object will cache the result.

`os.DirEntry` instances are not intended to be stored in long-lived data structures; if you know the file metadata has changed or if a long time has elapsed since calling `scandir()`, call `os.stat(entry.path)` to fetch up-to-date information.

Because the `os.DirEntry` methods can make operating system calls, they may also raise `OSError`. If you need very fine-grained control over errors, you can catch `OSError` when calling one of the `os.DirEntry` methods and handle as appropriate.

To be directly usable as a path-like object, `os.DirEntry` implements the `PathLike` interface.

Attributes and methods on a `os.DirEntry` instance are as follows:
name
The entry’s base filename, relative to the `scandir()` path argument.

The `name` attribute will be `bytes` if the `scandir()` path argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

path
The entry’s full path name: equivalent to `os.path.join(scandir_path, entry.name)` where `scandir_path` is the `scandir()` path argument. The path is only absolute if the `scandir()` path argument was absolute. If the `scandir()` path argument was a file descriptor, the `path` attribute is the same as the `name` attribute.

The `path` attribute will be `bytes` if the `scandir()` path argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

inode()
Return the inode number of the entry.

The result is cached on the `os.DirEntry` object. Use `os.stat(entry.path, follow_symlinks=False).st_ino` to fetch up-to-date information.

On the first, uncached call, a system call is required on Windows but not on Unix.

`is_dir(*, follow_symlinks=True)`
is_file(\*, follow_symlinks=True)

Return True if this entry is a file or a symbolic link pointing to a file; return False if the entry is or points to a directory or other non-file entry, or if it doesn’t exist anymore.

If follow_symlinks is False, return True only if this entry is a file (without following symlinks); return False if the entry is a directory or other non-file entry, or if it doesn’t exist anymore.
The result is cached on the `os.DirEntry` object. Caching, system calls made, and exceptions raised are as per `is_dir()`.

**is_symlink()**
Return `True` if this entry is a symbolic link (even if broken); return `False` if the entry points to a directory or any kind of file, or if it doesn’t exist anymore.

The result is cached on the `os.DirEntry` object. Call `os.path.islink()` to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`.

This method can raise `OSError`, such as `PermissionError`, but `FileNotFoundError` is caught and not raised.

**stat(*, follow_symlinks=True)**
Return a `stat_result` object for this entry. This method follows symbolic links by default; to stat a symbolic link add the `follow_symlinks=False` argument.

On Unix, this method always requires a system call. On Windows, it only requires a system call if `follow_symlinks` is `True` and the entry is a symbolic link.

On Windows, the `st_ino`, `st_dev` and `st_nlink` attributes of the `stat_result` are always set to zero. Call `os.stat()` to get these attributes.
The result is cached on the `os.DirEntry` object, with a separate cache for `follow_symlinks` `True` and `False`. Call `os.stat()` to fetch up-to-date information.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of `pathlib.Path`. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()` and `stat()` methods.

*New in version 3.5.*

*Changed in version 3.6:* Added support for the `PathLike` interface. Added support for `bytes` paths on Windows.

`os.stat(path, *, dir_fd=None, follow_symlinks=True)`

Get the status of a file or a file descriptor. Perform the equivalent of a `stat()` system call on the given path. `path` may be specified as either a string or bytes – directly or indirectly through the `PathLike` interface – or as an open file descriptor. Return a `stat_result` object.

This function normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use `lstat()`.

This function can support specifying a file descriptor and not following symlinks.

Example:

```python
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_
```python
st_nlink=1, st_uid=501, st_gid=501, st_size=264,
st_mtime=1297230027, st_ctime=1297230027)
```

```python
>>> statinfo.st_size
264
```

**See also:** `fstat()` and `lstat()` functions.

New in version 3.3: Added the `dir_fd` and `follow_symlinks` arguments, specifying a file descriptor instead of a path.

Changed in version 3.6: Accepts a path-like object.

```python
class os.stat_result
```

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of `os.stat()`, `os.fstat()` and `os.lstat()`.

Attributes:

**st_mode**

File mode: file type and file mode bits (permissions).

**st_ino**

Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index on Windows

**st_dev**

Identifier of the device on which this file resides.

**st_nlink**

Number of hard links.
**st_uid**
User identifier of the file owner.

**st_gid**
Group identifier of the file owner.

**st_size**
Size of the file in bytes, if it is a regular file or a symbolic link. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Timestamps:

**st_atime**
Time of most recent access expressed in seconds.

**st_mtime**
Time of most recent content modification expressed in seconds.

**st_ctime**
Platform dependent:

- the time of most recent metadata change on Unix,
- the time of creation on Windows, expressed in seconds.

**st_atime_ns**
Time of most recent access expressed in nanoseconds as an integer.

**st_mtime_ns**
Time of most recent content modification expressed in nanoseconds as an integer.

**st_ctime_ns**
Platform dependent:

- the time of most recent metadata change on Unix,
- the time of creation on Windows, expressed in nanoseconds as an integer.

Note: The exact meaning and resolution of the `st_atime`, `st_mtime`, and `st_ctime` attributes depend on the operating system and the file system. For example, on Windows systems using the FAT or FAT32 file systems, `st_mtime` has 2-second resolution, and `st_atime` has only 1-day resolution. See your operating system documentation for details. Similarly, although `st_atime_ns`, `st_mtime_ns`, and `st_ctime_ns` are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store `st_atime`, `st_mtime`, and `st_ctime` cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use `st_atime_ns`, `st_mtime_ns`, and `st_ctime_ns`.

On some Unix systems (such as Linux), the following attributes may also be available:

**st_blocks**

Number of 512-byte blocks allocated for file. This may be smaller than `st_size`/512 when the file has holes.

**st_blksize**

“Preferred” blocksize for efficient file system I/O. Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.

**st_rdev**
Type of device if an inode device.

**st_flags**
User defined flags for file.

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

**st_gen**
File generation number.

**st_birthtime**
Time of file creation.

On Solaris and derivatives, the following attributes may also be available:

**st_fstype**
String that uniquely identifies the type of the filesystem that contains the file.

On Mac OS systems, the following attributes may also be available:

**st_rsize**
Real size of the file.

**st_creator**
Creator of the file.

**st_type**
File type.
On Windows systems, the following attribute is also available:

**st_file_attributes**

Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_`* constants in the `stat` module.

The standard module `stat` defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a `stat_result` instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order `st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime`. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers.

*New in version 3.3:* Added the `st_atime_ns, st_mtime_ns, and st_ctime_ns` members.

*New in version 3.5:* Added the `st_file_attributes` member on Windows.

*Changed in version 3.5:* Windows now returns the file index as `st_ino` when available.

*New in version 3.7:* Added the `st_fstype` member to Solaris/derivatives.
Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Two module-level constants are defined for the `f_flag` attribute's bit-flags: if `ST_RDONLY` is set, the filesystem is mounted read-only, and if `ST_NOSUID` is set, the semantics of setuid/setgid bits are disabled or not supported.

Additional module-level constants are defined for GNU/glibc based systems. These are `ST_NODEV` (disallow access to device special files), `ST_NOEXEC` (disallow program execution), `ST_SYNCHRONOUS` (writes are synced at once), `ST_MANDLOCK` (allow mandatory locks on an FS), `ST_WRITE` (write on file/directory/symlink), `ST_APPEND` (append-only file), `ST_IMMUTABLE` (immutable file), `ST_NOATIME` (do not update access times), `ST_NODIRATIME` (do not update directory access times), `ST_RELATIME` (update atime relative to mtime/ctime).

This function can support specifying a file descriptor.

**Availability:** Unix.

*Changed in version 3.2:* The `ST_RDONLY` and `ST_NOSUID` constants were added.

*New in version 3.3:* Added support for specifying an open file descriptor for `path`.
Changed in version 3.4: The ST_NODEV, ST_NOEXEC, ST_SYNCHRONOUS, ST_MANDLOCK, ST_WRITE, ST_APPEND, ST_IMMUTABLE, ST_NOATIME, ST_NODIRATIME, and ST_RELATIME constants were added.

Changed in version 3.6: Accepts a path-like object.

New in version 3.7: Added f_fsid.

os.supports_dir_fd

A Set object indicating which functions in the os module permit use of their dir_fd parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support dir_fd always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its dir_fd parameter, use the in operator on supports_dir_fd. As an example, this expression determines whether the dir_fd parameter of os.stat() is locally available:

```python
os.stat in os.supports_dir_fd
```

Currently dir_fd parameters only work on Unix platforms; none of them work on Windows.

New in version 3.3.

os.supports_effective_ids

A Set object indicating which functions in the os module permit use of the effective_ids parameter for os.access(). If the local platform supports it, the collection will contain os.access(),
otherwise it will be empty.

To check whether you can use the `effective_ids` parameter for `os.access()`, use the `in` operator on `supports_effective_ids`, like so:

```python
os.access in os.supports_effective_ids
```

Currently `effective_ids` only works on Unix platforms; it does not work on Windows.

*New in version 3.3.*

**os.supports_fd**

A `Set` object indicating which functions in the `os` module permit specifying their `path` parameter as an open file descriptor. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency’s sakes, functions that support `fd` always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits specifying an open file descriptor for its `path` parameter, use the `in` operator on `supports_fd`. As an example, this expression determines whether `os.chdir()` accepts open file descriptors when called on your local platform:

```python
os.chdir in os.supports_fd
```

*New in version 3.3.*

**os.supports_follow_symlinks**
A `Set` object indicating which functions in the `os` module permit use of their `follow_symlinks` parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency’s sakes, functions that support `follow_symlinks` always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its `follow_symlinks` parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression determines whether the `follow_symlinks` parameter of `os.stat()` is locally available:

```python
os.stat in os.supports_follow_symlinks
```

*New in version 3.3.*

`os.symlink(src, dst, target_is_directory=False, *, dir_fd=None)`

Create a symbolic link pointing to `src` named `dst`.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if `target_is_directory` is `True` or a file symlink (the default) otherwise. On non-Windows platforms, `target_is_directory` is ignored.

Symbolic link support was introduced in Windows 6.0 (Vista). `symlink()` will raise a `NotImplementedError` on Windows versions earlier than 6.0.

This function can support `paths relative to directory descriptors`. 
Note: On Windows, the `SeCreateSymbolicLinkPrivilege` is required in order to successfully create symlinks. This privilege is not typically granted to regular users but is available to accounts which can escalate privileges to the administrator level. Either obtaining the privilege or running your application as an administrator are ways to successfully create symlinks. `OSError` is raised when the function is called by an unprivileged user.

Availability: Unix, Windows.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

New in version 3.3: Added the `dir_fd` argument, and now allow `target_is_directory` on non-Windows platforms.

Changed in version 3.6: Accepts a path-like object for `src` and `dst`.

`os.sync()`

Force write of everything to disk.

Availability: Unix.

New in version 3.3.

`os.truncate(path, length)`

Truncate the file corresponding to `path`, so that it is at most `length` bytes in size.

This function can support specifying a file descriptor.

Availability: Unix, Windows.
New in version 3.3.

Changed in version 3.5: Added support for Windows

Changed in version 3.6: Accepts a path-like object.

`os.unlink(path, *, dir_fd=None)`

Remove (delete) the file `path`. This function is semantically identical to `remove()`; the `unlink` name is its traditional Unix name. Please see the documentation for `remove()` for further information.

New in version 3.3: The `dir_fd` parameter.

Changed in version 3.6: Accepts a path-like object.

`os.utime(path, times=None, *, [ns, ]dir_fd=None, follow_symlinks=True)`

Set the access and modified times of the file specified by `path`.

`utime()` takes two optional parameters, `times` and `ns`. These specify the times set on `path` and are used as follows:

- If `ns` is specified, it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- If `times` is not `None`, it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.
- If `times` is `None` and `ns` is unspecified, this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both `times` and `ns`.
Whether a directory can be given for `path` depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. The best way to preserve exact times is to use the `st_atime_ns` and `st_mtime_ns` fields from the `os.stat()` result object with the `ns` parameter to `utime`.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

*New in version 3.3:* Added support for specifying an open file descriptor for `path`, and the `dir_fd`, `follow_symlinks`, and `ns` parameters.

*Changed in version 3.6:* Accepts a path-like object.

`os.walk(top, topdown=True, onerror=None, followlinks=False)`

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory `top` (including `top` itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

`dirpath` is a string, the path to the directory. `dirnames` is a list of the names of the subdirectories in `dirpath` (excluding `'.'` and `..'`). `filenames` is a list of the names of the non-directory files in `dirpath`. Note that the names in the lists contain no path components. To get a full path (which begins with `top`) to a file or directory in `dirpath`, do `os.path.join(dirpath, name)`.

If optional argument `topdown` is `True` or not specified, the triple for a directory is generated before the triples for any of its
subdirectories (directories are generated top-down). If \textit{topdown} is \texttt{False}, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of \textit{topdown}, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When \textit{topdown} is \texttt{True}, the caller can modify the \texttt{dirnames} list in-place (perhaps using \texttt{del} or slice assignment), and \texttt{walk()} will only recurse into the subdirectories whose names remain in \texttt{dirnames}; this can be used to prune the search, impose a specific order of visiting, or even to inform \texttt{walk()} about directories the caller creates or renames before it resumes \texttt{walk()} again. Modifying \texttt{dirnames} when \textit{topdown} is \texttt{False} has no effect on the behavior of the walk, because in bottom-up mode the directories in \texttt{dirnames} are generated before \texttt{dirpath} itself is generated.

By default, errors from the \texttt{scandir()} call are ignored. If optional argument \texttt{onerror} is specified, it should be a function; it will be called with one argument, an \texttt{OSError} instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the \texttt{filename} attribute of the exception object.

By default, \texttt{walk()} will not walk down into symbolic links that resolve to directories. Set \texttt{followlinks} to \texttt{True} to visit directories pointed to by symlinks, on systems that support them.

\begin{quote}
\textbf{Note:} Be aware that setting \texttt{followlinks} to \texttt{True} can lead to infinite recursion if a link points to a parent directory of itself. \texttt{walk()} does not keep track of the directories it visited already.
\end{quote}
Note: If you pass a relative pathname, don’t change the current working directory between resumptions of `walk()`. `walk()` never changes the current directory, and assumes that its caller doesn’t either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any CVS subdirectory:

```python
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email', topdown=True):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name))) for name in files,
          "bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example (simple implementation of `shutil.rmtree()`), walking the tree bottom-up is essential, `rmdir()` doesn’t allow deleting a directory before the directory is empty:

```python
# Delete everything reachable from the directory (top), assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/',
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```
*Changed in version 3.5:* This function now calls `os.scandir()` instead of `os.listdir()`, making it faster by reducing the number of calls to `os.stat()`.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.fwalk(top='.', topdown=True, onerror=None, *,
follow_symlinks=False, dir_fd=None)
```

This behaves exactly like `walk()`, except that it yields a 4-tuple `(dirpath, dirnames, filenames, dirfd)`, and it supports `dir_fd`.

`dirpath`, `dirnames` and `filenames` are identical to `walk()` output, and `dirfd` is a file descriptor referring to the directory `dirpath`.

This function always supports paths relative to directory descriptors and not following symlinks. Note however that, unlike other functions, the `fwalk()` default value for `follow_symlinks` is `False`.

**Note:** Since `fwalk()` yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with `dup()`) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any CVS subdirectory:

```python
import os
for root, dirs, files, rootfd in os.fwalk('python Lib/email'
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size
                 end="")
    print("bytes in", len(files), "non-directory")
```
In the next example, walking the tree bottom-up is essential: `rmdir()` doesn't allow deleting a directory before the directory is empty:

```python
if 'CVS' in dirs:
    dirs.remove('CVS')  # don't visit CVS directory
```

# Delete everything reachable from the directory (e.g., files, subdirectories) assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top is '/', it could delete all your disk files.

```python
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

**Availability:** Unix.

*New in version 3.3.*

*Changed in version 3.6:* Accepts a *path-like object*.

*Changed in version 3.7:* Added support for *bytes* paths.

**Linux extended attributes**

*New in version 3.3.*

These functions are all available on Linux only.

```python
os.getxattr(path, attribute, *, follow_symlinks=True)
```
Return the value of the extended filesystem attribute `attribute` for `path`. `attribute` can be bytes or str (directly or indirectly through the `PathLike` interface). If it is str, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

*Changed in version 3.6:* Accepts a path-like object for `path` and `attribute`.

```python
os.listxattr(path=None, *, follow_symlinks=True)
```

Return a list of the extended filesystem attributes on `path`. The attributes in the list are represented as strings decoded with the filesystem encoding. If `path` is None, `listxattr()` will examine the current directory.

This function can support specifying a file descriptor and not following symlinks.

*Changed in version 3.6:* Accepts a path-like object.

```python
os.removexattr(path, attribute, *, follow_symlinks=True)
```

Removes the extended filesystem attribute `attribute` from `path`. `attribute` should be bytes or str (directly or indirectly through the `PathLike` interface). If it is a string, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

*Changed in version 3.6:* Accepts a path-like object for `path` and `attribute`. 
**os.setxattr**(*path, attribute, value, flags=0, *, follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the filesystem encoding. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, EEXISTS will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and ENODATA will be raised.

This function can support **specifying a file descriptor** and **not following symlinks**.

**Note:** A bug in Linux kernel versions less than 2.6.39 caused the *flags* argument to be ignored on some filesystems.

*Changed in version 3.6:* Accepts a *path-like object* for *path* and *attribute*.

**os.XATTR_SIZE_MAX**

The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

**os.XATTR_CREATE**

This is a possible value for the *flags* argument in *setxattr()*. It indicates the operation must create an attribute.

**os.XATTR_REPLACE**

This is a possible value for the *flags* argument in *setxattr()*. It indicates the operation must replace an existing attribute.
Process Management

These functions may be used to create and manage processes.

The various exec* functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the argv[0] passed to a program’s main(). For example, os.execv('/bin/echo', ['foo', 'bar']) will only print bar on standard output; foo will seem to be ignored.

os.abort()
Generate a SIGABRT signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that calling this function will not call the Python signal handler registered for SIGABRT with signal.signal().

os.execl(path, arg0, arg1,...)

os.execl(path, arg0, arg1,..., env)

os.execlp(file, arg0, arg1,...)

os.execlpe(file, arg0, arg1,..., env)

os.execv(path, args)

os.execve(path, args, env)

os.execvp(file, args)

os.execvpe(file, args, env)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as OSError exceptions.
The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or `os.fsync()` before calling an `exec*` function.

The “l” and “v” variants of the `exec*` functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `exec1*()` functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the `args` parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a “p” near the end (`exec1p()`, `exec1pe()`, `execvp()`, and `execvpe()`) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, `exec1()`, `execle()`, `execv()`, and `execve()`, will not use the PATH variable to locate the executable; `path` must contain an appropriate absolute or relative path.

For `execle()`, `exec1pe()`, `execve()`, and `execvpe()` (note that these all end in “e”), the `env` parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process’ environment); the functions `exec1()`, `exec1p()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of
the current process.

For `execve()` on some platforms, `path` may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

**Availability:** Unix, Windows.

*New in version 3.3:* Added support for specifying an open file descriptor for `path` for `execve()`.

*Changed in version 3.6:* Accepts a path-like object.

```
``` os._exit(n)
Exit the process with status `n`, without calling cleanup handlers, flushing stdio buffers, etc.

**Note:** The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

```
``` The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server’s external command delivery program.

**Note:** Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

```
``` os.EX_OK
Exit code that means no error occurred.

**Availability**: Unix.

**os. EX_USAGE**
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

**Availability**: Unix.

**os. EX_DATAERR**
Exit code that means the input data was incorrect.

**Availability**: Unix.

**os. EX_NOINPUT**
Exit code that means an input file did not exist or was not readable.

**Availability**: Unix.

**os. EX_NOUSER**
Exit code that means a specified user did not exist.

**Availability**: Unix.

**os. EX_NOHOST**
Exit code that means a specified host did not exist.

**Availability**: Unix.

**os. EX_UNAVAILABLE**
Exit code that means that a required service is unavailable.

**Availability**: Unix.
os. **EXSOFTWARE**
Exit code that means an internal software error was detected.

**Availability**: Unix.

os. **EXOSERR**
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

**Availability**: Unix.

os. **EXOSFILE**
Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

**Availability**: Unix.

os. **EXCANTCREAT**
Exit code that means a user specified output file could not be created.

**Availability**: Unix.

os. **EXIOERR**
Exit code that means that an error occurred while doing I/O on some file.

**Availability**: Unix.

os. **EXTEMPFAIL**
Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.
**Availability:** Unix.

**os.EX_PROTOCOL**
Exit code that means that a protocol exchange was illegal, invalid, or not understood.

**Availability:** Unix.

**os.EX_NOPERM**
Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

**Availability:** Unix.

**os.EX_CONFIG**
Exit code that means that some kind of configuration error occurred.

**Availability:** Unix.

**os.EX_NOTFOUND**
Exit code that means something like “an entry was not found”.

**Availability:** Unix.

**os.fork()**
Fork a child process. Return 0 in the child and the child’s process id in the parent. If an error occurs OSError is raised.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using fork() from a thread.

**Warning:** See ssl for applications that use the SSL module with fork().
os.**forkpty()**
Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of \((\text{pid}, \text{fd})\), where \(\text{pid}\) is 0 in the child, the new child's process id in the parent, and \(\text{fd}\) is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the `pty` module. If an error occurs `OSError` is raised.

**Availability:** some flavors of Unix.

os.**kill(pid, sig)**
Send signal \(\text{sig}\) to the process \(\text{pid}\). Constants for the specific signals available on the host platform are defined in the `signal` module.

Windows: The `signal.CTRL_C_EVENT` and `signal.CTRL_BREAK_EVENT` signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for \(\text{sig}\) will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to \(\text{sig}\). The Windows version of `kill()` additionally takes process handles to be killed.

See also `signal.pthread_kill()`.

*New in version 3.2:* Windows support.

os.**killpg(pgid, sig)**
Send the signal \(\text{sig}\) to the process group \(\text{pgid}\).
Availability: Unix.

os.nice(increment)
Add increment to the process’s “niceness”. Return the new niceness.

Availability: Unix.

os.plock(op)
Lock program segments into memory. The value of op (defined in <sys/lock.h>) determines which segments are locked.

Availability: Unix.

os.popen(cmd, mode='r', buffering=-1)
Open a pipe to or from command cmd. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'. The buffering argument has the same meaning as the corresponding argument to the built-in open() function. The returned file object reads or writes text strings rather than bytes.

The close method returns None if the subprocess exited successfully, or the subprocess’s return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be -signal.SIGKILL if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

This is implemented using subprocess.Popen; see that class's
documentation for more powerful ways to manage and communicate with subprocesses.

```python
os.register_at_fork(*, before=None, after_in_parent=None, after_in_child=None)
```

Register callables to be executed when a new child process is forked using `os.fork()` or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- `before` is a function called before forking a child process.
- `after_in_parent` is a function called from the parent process after forking a child process.
- `after_in_child` is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical `subprocess` launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

**Availability:** Unix.

*New in version 3.7.*
os. `spawnl`(mode, path, ...)  
os. `spawnle`(mode, path, ..., env)  
os. `spawnlp`(mode, file, ...)  
os. `spawnlpe`(mode, file, ..., env)  
os. `spawnv`(mode, path, args)  
os. `spawnve`(mode, path, args, env)  
os. `spawnvp`(mode, file, args)  
os. `spawnvpe`(mode, file, args, env)

Execute the program path in a new process.

(Note that the subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If mode is P_NOWAIT, this function returns the process id of the new process; if mode is P_WAIT, returns the process's exit code if it exits normally, or -signal, where signal is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the `waitpid()` function.

The “l” and “v” variants of the spawn* functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the spawnl*() functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the args parameter. In either case, the arguments to the child process must start with the name of the command being run.
The variants which include a second “p” near the end (spawnlp(), spawnlpe(), spawnvp(), and spawnvpe()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the spawn*e variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, spawnl(), spawnle(), spawnv(), and spawnve(), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path.

For spawnle(), spawnlpe(), spawnve(), and spawnvpe() (note that these all end in “e”), the env parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process’ environment); the functions spawnl(), spawnlp(), spawnv(), and spawnvp() all cause the new process to inherit the environment of the current process. Note that keys and values in the env dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of 127.

As an example, the following calls to spawnlp() and spawnvpe() are equivalent:

```python
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '
L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. spawnlp(), spawnlpe(), spawnvp() and spawnvpe() are not available on Windows.
spawnle() and spawnve() are not thread-safe on Windows; we advise you to use the subprocess module instead.

*Changed in version 3.6:* Accepts a path-like object.

**os.P_NOWAIT**

**os.P_NOWAITO**

Possible values for the *mode* parameter to the spawn* family of functions. If either of these values is given, the spawn*() functions will return as soon as the new process has been created, with the process id as the return value.

**Availability:** Unix, Windows.

**os.P_WAIT**

Possible value for the *mode* parameter to the spawn* family of functions. If this is given as *mode*, the spawn*() functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or -signal if a signal kills the process.

**Availability:** Unix, Windows.

**os.P_DETACH**

**os.P_OVERLAY**

Possible values for the *mode* parameter to the spawn* family of functions. These are less portable than those listed above. P_DETACH is similar to P_NOWAIT, but the new process is detached from the console of the calling process. If P_OVERLAY is used, the current process will be replaced; the spawn* function will not return.

**Availability:** Windows.
os.startfile(path[, operation])

Start a file with its associated application.

When operation is not specified or 'open', this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the start command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another operation is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are 'print' and 'edit' (to be used on files) as well as 'explore' and 'find' (to be used on directories).

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The path parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash ('/'); the underlying Win32 ShellExecute() function doesn't work if it is. Use the os.path.normpath() function to ensure that the path is properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 ShellExecute() function is not resolved until this function is first called. If the function cannot be resolved, NotImplementedError will be raised.

Availability: Windows.

os.system(command)
Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations. Changes to sys.stdin, etc. are not reflected in the environment of the executed command. If command generates any output, it will be sent to the interpreter standard output stream.

On Unix, the return value is the exit status of the process encoded in the format specified for wait(). Note that POSIX does not specify the meaning of the return value of the C system() function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running command. The shell is given by the Windows environment variable COMSPEC: it is usually cmd.exe, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the Replacing Older Functions with the subprocess Module section in the subprocess documentation for some helpful recipes.

Availability: Unix, Windows.

os.times()

Returns the current global process times. The return value is an object with five attributes:

- user - user time
- system - system time
- children_user - user time of all child processes
- children_system - system time of all child processes
- elapsed - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing user, system, children_user, children_system, and elapsed in that order.

See the Unix manual page times(2) or the corresponding Windows Platform API documentation. On Windows, only user and system are known; the other attributes are zero.

**Availability**: Unix, Windows.

*Changed in version 3.3:* Return type changed from a tuple to a tuple-like object with named attributes.

**os.wait()**

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

**Availability**: Unix.

**os.waitid(idtype, id, options)**

Wait for the completion of one or more child processes. idtype can be P_PID, P_Pgid or P_ALL. id specifies the pid to wait on. options is constructed from the ORing of one or more of WEXITED, WSTOPPED or WCONTINUED and additionally may be ORed with WNOHANG or WNOWAIT. The return value is an object representing the data contained in the siginfo_t structure, namely: si_pid, si_uid, si_signo, si_status, si_code or None if WNOHANG
is specified and there are no children in a waitable state.

**Availability:** Unix.

*New in version 3.3.*

**os.P_PID**
**os.P_PGID**
**os.P_ALL**

These are the possible values for `idtype` in `waitid()`. They affect how `id` is interpreted.

**Availability:** Unix.

*New in version 3.3.*

**os.WEXITED**
**os.WSTOPPED**
**os.WNOWAIT**

Flags that can be used in `options` in `waitid()` that specify what child signal to wait for.

**Availability:** Unix.

*New in version 3.3.*

**os.CLD_EXITED**
**os.CLD_DUMPED**
**os.CLD_TRAPPED**
**os.CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by `waitid()`.

**Availability:** Unix.
New in version 3.3.

**os.waitpid(pid, options)**

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id `pid`, and return a tuple containing its process id and exit status indication (encoded as for `wait()`). The semantics of the call are affected by the value of the integer `options`, which should be 0 for normal operation.

If `pid` is greater than 0, `waitpid()` requests status information for that specific process. If `pid` is 0, the request is for the status of any child in the process group of the current process. If `pid` is -1, the request pertains to any child of the current process. If `pid` is less than -1, status is requested for any process in the process group `-pid` (the absolute value of `pid`).

An **OSError** is raised with the value of errno when the syscall returns -1.

On Windows: Wait for completion of a process given by process handle `pid`, and return a tuple containing `pid`, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A `pid` less than or equal to 0 has no special meaning on Windows, and raises an exception. The value of integer `options` has no effect. `pid` can refer to any process whose id is known, not necessarily a child process. The `spawn*` functions called with **P_NOWAIT** return suitable process handles.

**Changed in version 3.5:** If the system call is interrupted and the signal handler does not raise an exception, the function now retries.
the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

**os. `wait3(options)`**
Similar to `waitpid()`, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The option argument is the same as that provided to `waitpid()` and `wait4()`.

*Availability:* Unix.

**os. `wait4(pid, options)`**
Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

*Availability:* Unix.

**os. `WNOHANG`**
The option for `waitpid()` to return immediately if no child process status is available immediately. The function returns `(0, 0)` in this case.

*Availability:* Unix.

**os. `WCONTINUED`**
This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported.
Availability: some Unix systems.

**os.** `WUNTRACED`

This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped.

Availability: Unix.

The following functions take a process status code as returned by `system()`, `wait()`, or `waitpid()` as a parameter. They may be used to determine the disposition of a process.

**os.** `WCOREDUMP(status)`

Return `True` if a core dump was generated for the process, otherwise return `False`.

Availability: Unix.

**os.** `WIFCONTINUED(status)`

Return `True` if the process has been continued from a job control stop, otherwise return `False`.

Availability: Unix.

**os.** `WIFSTOPPED(status)`

Return `True` if the process has been stopped, otherwise return `False`.

Availability: Unix.

**os.** `WIFSIGNALED(status)`

Return `True` if the process exited due to a signal, otherwise return
Return **True** if the process exited using the `exit(2)` system call, otherwise return **False**.

**Availability:** Unix.

**os. WIFEXITED**(status)

If **WIFEXITED**(status) is true, return the integer parameter to the `exit(2)` system call. Otherwise, the return value is meaningless.

**Availability:** Unix.

**os. WEXITSTATUS**(status)

Return the signal which caused the process to stop.

**Availability:** Unix.

**os. WSTOPSIG**(status)

Return the signal which caused the process to exit.

**Availability:** Unix.
Interface to the scheduler

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages.

*New in version 3.3.*

The following scheduling policies are exposed if they are supported by the operating system.

- **os.SCHED_OTHER**
  The default scheduling policy.

- **os.SCHED_BATCH**
  Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

- **os.SCHED_IDLE**
  Scheduling policy for extremely low priority background tasks.

- **os.SCHED_SPORADIC**
  Scheduling policy for sporadic server programs.

- **os.SCHED_FIFO**
  A First In First Out scheduling policy.

- **os.SCHED_RR**
  A round-robin scheduling policy.

- **os.SCHED_RESET_ON_FORK**
  This flag can be OR’ed with any other scheduling policy. When a process with this flag set forks, its child’s scheduling policy and
priority are reset to the default.

class os.sched_param(sched_priority)
   This class represents tunable scheduling parameters used in sched_setparam(), sched_setscheduler(), and sched_getparam(). It is immutable.

   At the moment, there is only one possible parameter:

   sched_priority
   The scheduling priority for a scheduling policy.

os.sched_get_priority_min(policy)
   Get the minimum priority value for policy. policy is one of the scheduling policy constants above.

os.sched_get_priority_max(policy)
   Get the maximum priority value for policy. policy is one of the scheduling policy constants above.

os.sched_setscheduler(pid, policy, param)
   Set the scheduling policy for the process with PID pid. A pid of 0 means the calling process. policy is one of the scheduling policy constants above. param is a sched_param instance.

os.sched_getscheduler(pid)
   Return the scheduling policy for the process with PID pid. A pid of 0 means the calling process. The result is one of the scheduling policy constants above.

os.sched_setparam(pid, param)
   Set a scheduling parameters for the process with PID pid. A pid of 0 means the calling process. param is a sched_param instance.
os. `sched_getparam(pid)`
Return the scheduling parameters as a `sched_param` instance for the process with PID `pid`. A `pid` of 0 means the calling process.

os. `sched_rr_get_interval(pid)`
Return the round-robin quantum in seconds for the process with PID `pid`. A `pid` of 0 means the calling process.

os. `sched_yield()`
Voluntarily relinquish the CPU.

os. `sched_setaffinity(pid, mask)`
Restrict the process with PID `pid` (or the current process if zero) to a set of CPUs. `mask` is an iterable of integers representing the set of CPUs to which the process should be restricted.

os. `sched_getaffinity(pid)`
Return the set of CPUs the process with PID `pid` (or the current process if zero) is restricted to.
Miscellaneous System Information

**os.confstr(name)**

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn’t defined, `None` is returned.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `confstr_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

*Availability*: Unix.

**os.confstr_names**

Dictionary mapping names accepted by `confstr()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

*Availability*: Unix.

**os.cpu_count()**

Return the number of CPUs in the system. Returns `None` if
This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with `len(os.sched_getaffinity(0))`.

*New in version 3.4.*

**os. `getloadavg()`**

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises `OSError` if the load average was unobtainable.

*Availability:* Unix.

**os. `sysconf(name)`**

Return integer-valued system configuration values. If the configuration value specified by `name` isn’t defined, `-1` is returned. The comments regarding the `name` parameter for `confstr()` apply here as well; the dictionary that provides information on the known names is given by `sysconf_names`.

*Availability:* Unix.

**os. `sysconf_names`**

Dictionary mapping names accepted by `sysconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

*Availability:* Unix.

The following data values are used to support path manipulation operations. These are defined for all platforms.
Higher-level operations on pathnames are defined in the `os.path` module.

**os.curdir**

The constant string used by the operating system to refer to the current directory. This is `.' for Windows and POSIX. Also available via `os.path`.

**os.pardir**

The constant string used by the operating system to refer to the parent directory. This is `..` for Windows and POSIX. Also available via `os.path`.

**os.sep**

The character used by the operating system to separate pathname components. This is `/` for POSIX and `\` for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use `os.path.split()` and `os.path.join()` — but it is occasionally useful. Also available via `os.path`.

**os.altsep**

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `/` on Windows systems where `sep` is a backslash. Also available via `os.path`.

**os.extsep**

The character which separates the base filename from the extension; for example, the `.' in `os.py`. Also available via `os.path`.

**os.pathsep**
The character conventionally used by the operating system to separate search path components (as in \texttt{PATH}), such as ':;' for POSIX or ';;' for Windows. Also available via \texttt{os.path}.

\texttt{os.defpath}

The default search path used by \texttt{exec*p*} and \texttt{spawn*p*} if the environment doesn't have a 'PATH' key. Also available via \texttt{os.path}.

\texttt{os.linesep}

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as '\n' for POSIX, or multiple characters, for example, '\r\n' for Windows. Do not use \texttt{os.linesep} as a line terminator when writing files opened in text mode (the default); use a single '\n' instead, on all platforms.

\texttt{os.devnull}

The file path of the null device. For example: '/dev/null' for POSIX, 'nul' for Windows. Also available via \texttt{os.path}.

\texttt{os.RTLD_LAZY}
\texttt{os.RTLD_NOW}
\texttt{os.RTLD_GLOBAL}
\texttt{os.RTLD_LOCAL}
\texttt{os.RTLD_NODELETE}
\texttt{os.RTLD_NOLOAD}
\texttt{os.RTLD_DEEPBIND}

Flags for use with the \texttt{setdlopenflags()} and \texttt{getdlopenflags()} functions. See the Unix manual page \texttt{dlopen(3)} for what the different flags mean.
New in version 3.3.
Random numbers

**os.getrandom(size, flags=0)**

Get up to `size` random bytes. The function can return less bytes than requested.

These bytes can be used to seed user-space random number generators or for cryptographic purposes.

`getrandom()` relies on entropy gathered from device drivers and other sources of environmental noise. Unnecessarily reading large quantities of data will have a negative impact on other users of the `/dev/random` and `/dev/urandom` devices.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: `os.GRND_RANDOM` and `GRND_NONBLOCK`.

See also the Linux getrandom() manual page.

**Availability**: Linux 3.17 and newer.

*New in version 3.6.*

**os.urandom(size)**

Return a string of `size` random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation.

On Linux, if the `getrandom()` syscall is available, it is used in
blocking mode: block until the system urandom entropy pool is initialized (128 bits of entropy are collected by the kernel). See the **PEP 524** for the rationale. On Linux, the `getrandom()` function can be used to get random bytes in non-blocking mode (using the `GRND_NONBLOCK` flag) or to poll until the system urandom entropy pool is initialized.

On a Unix-like system, random bytes are read from the `/dev/urandom` device. If the `/dev/urandom` device is not available or not readable, the `NotImplementedError` exception is raised.

On Windows, it will use `CryptGenRandom()`.

**See also:** The `secrets` module provides higher level functions. For an easy-to-use interface to the random number generator provided by your platform, please see `random.SystemRandom`.

*Changed in version 3.6.0:* On Linux, `getrandom()` is now used in blocking mode to increase the security.

*Changed in version 3.5.2:* On Linux, if the `getrandom()` syscall blocks (the urandom entropy pool is not initialized yet), fall back on reading `/dev/urandom`.

*Changed in version 3.5:* On Linux 3.17 and newer, the `getrandom()` syscall is now used when available. On OpenBSD 5.6 and newer, the C `getentropy()` function is now used. These functions avoid the usage of an internal file descriptor.

```python
os.GRND_NONBLOCK
```
By default, when reading from `/dev/random`, `getrandom()` blocks if no random bytes are available, and when reading from `/dev/urandom`, it blocks if the entropy pool has not yet been initialized.

If the `GRND_NONBLOCK` flag is set, then `getrandom()` does not block in these cases, but instead immediately raises `BlockingIOError`.

*New in version 3.6.*

**os. GRND_RANDOM**

If this bit is set, then random bytes are drawn from the `/dev/random` pool instead of the `/dev/urandom` pool.

*New in version 3.6.*
io — Core tools for working with streams

Source code: Lib/io.py
Overview

The `io` module provides Python’s main facilities for dealing with various types of I/O. There are three main types of I/O: text I/O, binary I/O and raw I/O. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are stream and file-like object.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a `str` object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a `bytes` object to the `write()` method of a text stream.

*Changed in version 3.3:* Operations that used to raise `IOError` now raise `OSError`, since `IOError` is now an alias of `OSError`.

Text I/O

Text I/O expects and produces `str` objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with `open()`, optionally
specifying an encoding:

```python
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as **StringIO** objects:

```python
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of **TextIOBase**.

## Binary I/O

Binary I/O (also called *buffered I/O*) expects **bytes-like objects** and produces **bytes** objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `b` in the mode string:

```python
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as **BytesIO** objects:

```python
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of **BufferedIOBase**.

Other library modules may provide additional ways to create text or
binary streams. See `socket.socket.makefile()` for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOWBase`. 
High-level Module Interface

**io.** `DEFAULT_BUFFER_SIZE`  
An int containing the default buffer size used by the module's buffered I/O classes. `open()` uses the file's blksize (as obtained by `os.stat()`) if possible.

**io.** `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`  
This is an alias for the builtin `open()` function.

**exception** `io.** BlockingIOError`  
This is a compatibility alias for the builtin `BlockingIOError` exception.

**exception** `io.** UnsupportedOperation`  
An exception inheriting `OSError` and `ValueError` that is raised when an unsupported operation is called on a stream.

In-memory streams

It is also possible to use a **str** or **bytes-like object** as a file for both reading and writing. For strings `StringIO` can be used like a file opened in text mode. `BytesIO` can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

**See also:**

`sys` contains the standard IO streams: `sys.stdin`, `sys.stdout`,
and `sys.stderr`.
Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First abstract base classes (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

**Note:** The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, `BufferedIOWBase` provides unoptimized implementations of `readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class `I0Base`. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise `UnsupportedOperation` if they do not support a given operation.

The `RawI0Base` ABC extends `I0Base`. It deals with the reading and writing of bytes to a stream. `FileI0` subclasses `RawI0Base` to provide an interface to files in the machine’s file system.

The `BufferedI0Base` ABC deals with buffering on a raw byte stream (`RawI0Base`). Its subclasses, `BufferedWriter`, `BufferedReader`, and `BufferedReaderRWPair` buffer streams that are readable, writable, and both readable and writable. `BufferedReaderRandom` provides a buffered interface to random access streams. Another `BufferedReaderI0Base` subclass, `BytesI0`, is a stream of in-memory bytes.
The `TextIOBase` ABC, another subclass of `I0Base`, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. `TextIOWrapper`, which extends it, is a buffered text interface to a buffered raw stream (`BufferedI0Base`). Finally, `StringI0` is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of `open()` are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the `io` module:

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits</th>
<th>Stub Methods</th>
<th>Mixin Methods and Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOBase</td>
<td></td>
<td>fileno, seek, and truncate</td>
<td>close, closed, <code>__enter__</code>, <code>__exit__</code>, flush, isatty, <code>__iter__</code>, <code>__next__</code>, readable, readline, readlines, seekable, <code>tell</code>, writable, and writelines</td>
</tr>
<tr>
<td>RawIOBase</td>
<td>IOBase</td>
<td>readinto and write</td>
<td>Inherited <code>I0Base</code> methods, read, and readall</td>
</tr>
<tr>
<td>BufferedIOBase</td>
<td>IOBase</td>
<td>detach, read, read1, and write</td>
<td>Inherited <code>I0Base</code> methods, readinto, and readinto1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>detach</td>
<td>Inherited <code>I0Base</code> methods,</td>
</tr>
</tbody>
</table>
I/O Base Classes

class io.IOBase

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though IOBase does not declare read(), readinto(), or write() because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a ValueError (or UnsupportedOperation) when operations they do not support are called.

The basic type used for binary data read from or written to a file is bytes. Other bytes-like objects are accepted as method arguments too. In some cases, such as readinto(), a writable object such as bytearray is required. Text I/O classes work with str data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise ValueError in this case.

IOBase (and its subclasses) supports the iterator protocol,
meaning that an `IOBase` object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See `readline()` below.

`IOBase` is also a context manager and therefore supports the `with` statement. In this example, `file` is closed after the `with` statement's suite is finished—even if an exception occurs:

```python
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

`IOBase` provides these data attributes and methods:

**close()**
Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

**closed**
`True` if the stream is closed.

**fileno()**
Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError` is raised if the IO object does not use a file descriptor.

**flush()**
Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.
isatty()
Return True if the stream is interactive (i.e., connected to a terminal/tty device).

readable()
Return True if the stream can be read from. If False, read() will raise OSError.

readline(size=-1)
Read and return one line from the stream. If size is specified, at most size bytes will be read.

The line terminator is always b'\n' for binary files; for text files, the newline argument to open() can be used to select the line terminator(s) recognized.

readlines(hint=-1)
Read and return a list of lines from the stream. hint can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds hint.

Note that it’s already possible to iterate on file objects using for line in file: ... without calling file.readlines().

seek(offset[, whence])
Change the stream position to the given byte offset. offset is interpreted relative to the position indicated by whence. The default value for whence is SEEK_SET. Values for whence are:

- SEEK_SET or 0 – start of the stream (the default); offset
should be zero or positive
  • SEEK_CUR or 1 – current stream position; offset may be negative
  • SEEK_END or 2 – end of the stream; offset is usually negative

Return the new absolute position.

*New in version 3.1:* The `SEEK_*` constants.

*New in version 3.3:* Some operating systems could support additional values, like os.SEEK_HOLE or os.SEEK_DATA. The valid values for a file could depend on it being open in text or binary mode.

**seekable()**

Return `True` if the stream supports random access. If `False`, `seek()`, `tell()` and `truncate()` will raise `OSError`.

**tell()**

Return the current stream position.

**truncate(size=None)**

Resize the stream to the given `size` in bytes (or the current position if `size` is not specified). The current stream position isn’t changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

*Changed in version 3.5:* Windows will now zero-fill files when extending.

**writable()**
Return True if the stream supports writing. If False, write() and truncate() will raise OSError.

writelines(lines)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. IOBase provides a default implementation of this method that calls the instance's close() method.

class io.RawIOBase
Base class for raw binary I/O. It inherits IOBase. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from IOBase, RawIOBase provides the following methods:

read(size=-1)
Read up to size bytes from the object and return them. As a convenience, if size is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than size bytes may be returned if the operating system call returns fewer than size bytes.

If 0 bytes are returned, and size was not 0, this indicates end of
file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to `readall()` and `readinto()`.

**readall()**
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

**readinto(b)**
Read bytes into a pre-allocated, writable `bytes-like object b`, and return the number of bytes read. If the object is in non-blocking mode and no bytes are available, `None` is returned.

**write(b)**
Write the given `bytes-like object, b`, to the underlying raw stream, and return the number of bytes written. This can be less than the length of `b` in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate `b` after this method returns, so the implementation should only access `b` during the method call.

**class io.BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits `I0Base`. There is no public constructor.

The main difference with `RawIOBase` is that methods `read()`, `readinto()` and `write()` will try (respectively) to read as much input as requested or to consume all given output, at the expense
of making perhaps more than one system call.

In addition, those methods can raise `BlockingIOError` if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their `RawIOBase` counterparts, they will never return `None`.

Besides, the `read()` method does not have a default implementation that defers to `readinto()`.

A typical `BufferedIOBase` implementation should not inherit from a `RawIOBase` implementation, but wrap one, like `BufferedWriter` and `BufferedReader` do.

`BufferedIOBase` provides or overrides these methods and attribute in addition to those from `I0Base`:

`raw`

The underlying raw stream (a `RawIOBase` instance) that `BufferedIOBase` deals with. This is not part of the `BufferedIOBase` API and may not exist on some implementations.

`detach()`

Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperation`.
**New in version 3.1.**

**read**(size=-1)

Read and return up to size bytes. If the argument is omitted, None, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A BlockingIOError is raised if the underlying raw stream is in non-blocking-mode, and has no data available at the moment.

**read1**(size)

Read and return up to size bytes, with at most one call to the underlying raw stream’s read() (or readinto()) method. This can be useful if you are implementing your own buffering on top of a BufferedIOBase object.

If size is -1 (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

**readinto**(b)

Read bytes into a pre-allocated, writable bytes-like object b and return the number of bytes read.

Like read(), multiple reads may be issued to the underlying raw stream, unless the latter is interactive.
A **BlockingIOError** is raised if the underlying raw stream is in non-blocking-mode, and has no data available at the moment.

**readinto1(b)**
Read bytes into a pre-allocated, writable bytes-like object `b`, using at most one call to the underlying raw stream’s `read()` (or `readinto()`) method. Return the number of bytes read.

A **BlockingIOError** is raised if the underlying raw stream is in non-blocking-mode, and has no data available at the moment.

*New in version 3.5.*

**write(b)**
Write the given bytes-like object, `b`, and return the number of bytes written (always equal to the length of `b` in bytes, since if the write fails an **OSError** will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a **BlockingIOError** is raised if the data needed to be written to the raw stream but it couldn’t accept all the data without blocking.

The caller may release or mutate `b` after this method returns, so the implementation should only access `b` during the method call.

**Raw File I/O**
class `io.FileIO(name, mode='r', closefd=True, opener=None)`

`FileIO` represents an OS-level file containing bytes data. It implements the `RawIOBase` interface (and therefore the `IOBase` interface, too).

The `name` can be one of two things:

- a character string or `bytes` object representing the path to the file which will be opened. In this case `closefd` must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting `FileIO` object will give access. When the `FileIO` object is closed this `fd` will be closed as well, unless `closefd` is set to `False`.

The `mode` can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn’t exist when opened for writing or appending; it will be truncated when opened for writing. `FileExistsError` will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(name, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).
The newly created file is **non-inheritable**.

See the `open()` built-in function for examples on using the `opener` parameter.

*Changed in version 3.3:* The `opener` parameter was added. The `x` mode was added.

*Changed in version 3.4:* The file is now non-inheritable.

In addition to the attributes and methods from `IOBase` and `RawIOBase`, `FileIO` provides the following data attributes:

**mode**

The mode as given in the constructor.

**name**

The file name. This is the file descriptor of the file when no name is given in the constructor.

### Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

```python
class io.BytesIO([initial_bytes])
```

A stream implementation using an in-memory bytes buffer. It inherits `BufferedIOBase`. The buffer is discarded when the `close()` method is called.

The optional argument `initial_bytes` is a bytes-like object that contains initial data.
**BytesIO** provides or overrides these methods in addition to those from **BufferedIOBase** and **I0Base**:

**getbuffer()**
Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```python
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

**Note:** As long as the view exists, the **BytesIO** object cannot be resized or closed.

*New in version 3.2.*

**getvalue()**
Return bytes containing the entire contents of the buffer.

**read1([size])**
In **BytesIO**, this is the same as **read()**.

*Changed in version 3.7:* The size argument is now optional.

**readinto1(b)**
In **BytesIO**, this is the same as **readinto()**.

*New in version 3.5.*

```python
class io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```
A buffer providing higher-level access to a readable, sequential RawIOBase object. It inherits BufferedReader. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a BufferedReader for the given readable raw stream and buffer_size. If buffer_size is omitted, DEFAULT_BUFFER_SIZE is used.

BufferedReader provides or overrides these methods in addition to those from BufferedReader and I0Base:

**peek([size])**
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

**read([size])**
Read and return size bytes, or if size is not given or negative, until EOF or if the read call would block in non-blocking mode.

**read1([size])**
Read and return up to size bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

*Changed in version 3.7:* The size argument is now optional.

```
class io.BufferedWriter(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```
A buffer providing higher-level access to a writeable, sequential RawIOBase object. It inherits BufferedWriter. When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying RawIOBase object under various conditions, including:

- when the buffer gets too small for all pending data;
- when flush() is called;
- when a seek() is requested (for BufferedRandom objects);
- when the BufferedWriter object is closed or destroyed.

The constructor creates a BufferedWriter for the given writeable raw stream. If the buffer_size is not given, it defaults to DEFAULT_BUFFER_SIZE.

BufferedWriter provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

**flush()**
Force bytes held in the buffer into the raw stream. A BlockingIOError should be raised if the raw stream blocks.

**write(b)**
Write the bytes-like object, b, and return the number of bytes written. When in non-blocking mode, a BlockingIOError is raised if the buffer needs to be written out but the raw stream blocks.

```
class io.BufferedRandom(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```
A buffered interface to random access streams. It inherits BufferedReader and BufferedWriter, and further supports seek() and tell() functionality.
The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

BufferedReader is capable of anything BufferedReader or BufferedWriter can do.

class `io.BufferedRWPair(reader, writer, buffer_size=DEFAULT_BUFFER_SIZE)`

A buffered I/O object combining two unidirectional RawI/OBase objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits BufferedIOBase.

reader and writer are RawI/OBase objects that are readable and writeable respectively. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

BufferedReader implements all of BufferedIOBase’s methods except for detach(), which raises UnsupportedOperation.

**Warning:** BufferedReader does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use BufferedReader instead.

Text I/O

class `io.TextIOBase`

Base class for text streams. This class provides a character and line based interface to stream I/O. There is no readinto()
method because Python’s character strings are immutable. It inherits `IOBase`. There is no public constructor.

`TextIOWrapper` provides or overrides these data attributes and methods in addition to those from `IOBase`:

**encoding**

The name of the encoding used to decode the stream’s bytes into strings, and to encode strings into bytes.

**errors**

The error setting of the decoder or encoder.

**newlines**

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

**buffer**

The underlying binary buffer (a `BufferedIOBase` instance) that `TextIOWrapper` deals with. This is not part of the `TextIOWrapper` API and may not exist in some implementations.

**detach()**

Separate the underlying binary buffer from the `TextIOWrapper` and return it.

After the underlying buffer has been detached, the `TextIOWrapper` is in an unusable state.

Some `TextIOWrapper` implementations, like `StringIO`, may not have the concept of an underlying buffer and calling this method will raise `UnsupportedOperationException`.
New in version 3.1.

**read**(size=-1)
Read and return at most size characters from the stream as a single str. If size is negative or None, reads until EOF.

**readline**(size=-1)
Read until newline or EOF and return a single str. If the stream is already at EOF, an empty string is returned.

If size is specified, at most size characters will be read.

**seek**(offset[, whence])
Change the stream position to the given offset. Behaviour depends on the whence parameter. The default value for whence is SEEK_SET.

- SEEK_SET or 0: seek from the start of the stream (the default); offset must either be a number returned by TextIOBase.tell(), or zero. Any other offset value produces undefined behaviour.
- SEEK_CUR or 1: “seek” to the current position; offset must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; offset must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

New in version 3.1: The SEEK_* constants.

**tell()**
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the
underlying binary storage.

**write(s)**

Write the string s to the stream and return the number of characters written.

```python
class io.TextIOWrapper(buffer, encoding=None, errors=None, newline=None, line_buffering=False, write_through=False)
```

A buffered text stream over a **BufferedIOBase** binary stream. It inherits **TextIOWrapper**.

- **encoding** gives the name of the encoding that the stream will be decoded or encoded with. It defaults to **locale.getpreferredencoding(False)**.
- **errors** is an optional string that specifies how encoding and decoding errors are to be handled. Pass **'strict'** to raise a **ValueError** exception if there is an encoding error (the default of **None** has the same effect), or pass **'ignore'** to ignore errors. (Note that ignoring encoding errors can lead to data loss.) **'replace'** causes a replacement marker (such as '?'') to be inserted where there is malformed data. **'backslashreplace'** causes malformed data to be replaced by a backslashed escape sequence. When writing, **'xmlcharrefreplace'** (replace with the appropriate XML character reference) or **'namereplace'** (replace with \N{...} escape sequences) can be used. Any other error handling name that has been registered with **codecs.register_error()** is also valid.
- **newline** controls how line endings are handled. It can be **None**, '', '\n', '\r', and '\r\n'. It works as follows:
• When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `\n`, `\r`, or `\r\n`, and these are translated into `\n` before being returned to the caller. If it is `'`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

• When writing output to the stream, if `newline` is `None`, any `\n` characters written are translated to the system default line separator, `os.linesep`. If `newline` is `'` or `\n`, no translation takes place. If `newline` is any of the other legal values, any `\n` characters written are translated to the given string.

If `line_buffering` is `True`, `flush()` is implied when a call to `write` contains a newline character or a carriage return.

If `write_through` is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the `TextIOWrapper` object is immediately handled to its underlying binary buffer.

*Changed in version 3.3:* The `write_through` argument has been added.

*Changed in version 3.3:* The default `encoding` is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don’t change temporary the locale encoding using `locale.setlocale()`, use the current locale encoding instead of the user preferred encoding.

`TextIOWrapper` provides these members in addition to those of `TextIOBase` and its parents:
**line_buffering**

Whether line buffering is enabled.

**write_through**

Whether writes are passed immediately to the underlying binary buffer.

*New in version 3.7.*

**reconfigure**(*[, encoding][, errors][, newline][, line_buffering][, write_through]*)

Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except *errors='strict* is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

*New in version 3.7.*

**class io.StringIO**(initial_value='', newline='
')

An in-memory stream for text I/O. The text buffer is discarded when the *close()* method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by *write()* . The stream is positioned at the start of the buffer.
The *newline* argument works like that of *TextIOWrapper*. The default is to consider only \n characters as ends of lines and to do no newline translation. If *newline* is set to None, newlines are written as \n on all platforms, but universal newline decoding is still performed when reading.

**StringIO** provides this method in addition to those from *TextIOBase* and its parents:

**getvalue()**

Return a *str* containing the entire contents of the buffer. Newlines are decoded as if by *read()*, although the stream position is not changed.

Example usage:

```python
import io
output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n' contents = output.getvalue()

# Close object and discard memory buffer --
# ..getvalue() will now raise an exception.
output.close()
```

class *io.IncrementalNewlineDecoder*

A helper codec that decodes newlines for universal newlines mode. It inherits *codecs.IncrementalDecoder*. 
Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system’s unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, TextIOWrapper.tell() and TextIOWrapper.seek() are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to BytesIO.
Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a `signal` handler. If a thread tries to re-enter a buffered object which it is already accessing, a `RuntimeError` is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the `open()` function will wrap a buffered object inside a `TextIOWrapper`. This includes standard streams and therefore affects the built-in function `print()` as well.
time — Time access and conversions

This module provides various time-related functions. For related functionality, see also the datetime and calendar modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts, and is platform dependent. For Unix, the epoch is January 1, 1970, 00:00:00 (UTC). To find out what the epoch is on a given platform, look at `time.gmtime(0)`.

- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding leap seconds. Leap seconds are excluded from this total on all POSIX-compliant platforms.

- The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.

- **Year 2000 (Y2K) issues**: Python depends on the platform’s C library, which generally doesn’t have year 2000 issues, since all dates and times are represented internally as seconds since the
epoch. Function `strftime()` can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock “ticks” only 50 or 100 times a second.

- On the other hand, the precision of `time()` and `sleep()` is better than their Unix equivalents: times are expressed as floating point numbers, `time()` returns the most accurate time available (using Unix `gettimeofday()` where available), and `sleep()` will accept a time with a nonzero fraction (Unix `select()` is used to implement this, where available).

- The time value as returned by `gmtime()`, `localtime()`, and `strftime()`, and accepted by `asctime()`, `mktime()` and `strftime()`, is a sequence of 9 integers. The return values of `gmtime()`, `localtime()`, and `strftime()` also offer attribute
names for individual fields.

See \texttt{struct\_time} for a description of these objects.

\textit{Changed in version 3.3:} The \texttt{struct\_time} type was extended to provide the \texttt{tm\_gmtoff} and \texttt{tm\_zone} attributes when platform supports corresponding \texttt{struct tm} members.

\textit{Changed in version 3.6:} The \texttt{struct\_time} attributes \texttt{tm\_gmtoff} and \texttt{tm\_zone} are now available on all platforms.

- Use the following functions to convert between time representations:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds since the epoch</td>
<td>\texttt{struct_time in UTC}</td>
<td>\texttt{gmtime()}</td>
</tr>
<tr>
<td>seconds since the epoch</td>
<td>\texttt{struct_time in local time}</td>
<td>\texttt{localtime()}</td>
</tr>
<tr>
<td>\texttt{struct_time in UTC}</td>
<td>seconds since the epoch</td>
<td>\texttt{calendar.timegm()}</td>
</tr>
<tr>
<td>\texttt{struct_time in local time}</td>
<td>seconds since the epoch</td>
<td>\texttt{mktime()}</td>
</tr>
</tbody>
</table>
Functions

time.asctime([t])
Convert a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string of the following form: 'Sun Jun 20 23:21:05 1993'. If t is not provided, the current time as returned by localtime() is used. Locale information is not used by asctime().

**Note:** Unlike the C function of the same name, asctime() does not add a trailing newline.

time.clock()
On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of “processor time”, depends on that of the C function of the same name.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter(). The resolution is typically better than one microsecond.

*Deprecated since version 3.3:* The behaviour of this function depends on the platform: use perf_counter() or process_time() instead, depending on your requirements, to have a well defined behaviour.

ntime.pthread_getcpuclockid(thread_id)
Return the clk_id of the thread-specific CPU-time clock for the specified thread_id.
Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for `thread_id`.

**Warning:** Passing an invalid or expired `thread_id` may result in undefined behavior, such as segmentation fault.

**Availability:** Unix (see the man page for `pthread_getcpuclockid(3)` for further information).

*New in version 3.7.*

```python
def clock_getres(clk_id):
    Return the resolution (precision) of the specified clock `clk_id`. Refer to Clock ID Constants for a list of accepted values for `clk_id`.
```

**Availability:** Unix.

*New in version 3.3.*

```python
def clock_gettime(clk_id) -> float:
    Return the time of the specified clock `clk_id`. Refer to Clock ID Constants for a list of accepted values for `clk_id`.
```

**Availability:** Unix.

*New in version 3.3.*

```python
def clock_gettime_ns(clk_id) -> int:
    Similar to `clock_gettime()` but return time as nanoseconds.
```

**Availability:** Unix.

*New in version 3.7.*

```python
def clock_settime(clk_id, time:
    float):
```

```python
Set the time of the specified clock *clk_id*. Currently, `CLOCK_REALTIME` is the only accepted value for *clk_id*.

**Availability:** Unix.

*New in version 3.3.*

```python
time.clock_settime_ns(clk_id, time: int)
```

Similar to `clock_settime()` but set time with nanoseconds.

**Availability:** Unix.

*New in version 3.7.*

```python
time.ctime([secs])
```

Convert a time expressed in seconds since the epoch to a string representing local time. If secs is not provided or `None`, the current time as returned by `time()` is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `ctime()`.

```python
time.get_clock_info(name)
```

Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:

- `'clock'`: `time.clock()`
- `'monotonic'`: `time.monotonic()`
- `'perf_counter'`: `time.perf_counter()`
- `'process_time'`: `time.process_time()`
- `'thread_time'`: `time.thread_time()`
- `'time'`: `time.time()`

The result has the following attributes:
• adjustable: True if the clock can be changed automatically (e.g. by a NTP daemon) or manually by the system administrator, False otherwise
• implementation: The name of the underlying C function used to get the clock value. Refer to Clock ID Constants for possible values.
• monotonic: True if the clock cannot go backward, False otherwise
• resolution: The resolution of the clock in seconds (float)

New in version 3.3.

time.gmtime([secs])
Convert a time expressed in seconds since the epoch to a struct_time in UTC in which the dst flag is always zero. If secs is not provided or None, the current time as returned by time() is used. Fractions of a second are ignored. See above for a description of the struct_time object. See calendar.timegm() for the inverse of this function.

time.localtime([secs])
Like gmtime() but converts to local time. If secs is not provided or None, the current time as returned by time() is used. The dst flag is set to 1 when DST applies to the given time.

time.mktime(t)
This is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple (since the dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC. It returns a floating point number, for compatibility with time(). If the input value cannot be represented as a valid time, either OverflowError or ValueError will be raised
(which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

\texttt{time.monotonic()} → float

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

\textit{New in version 3.3.}

\textit{Changed in version 3.5}: The function is now always available and always system-wide.

\texttt{time.monotonic\_ns()} → int

Similar to \texttt{monotonic()}, but return time as nanoseconds.

\textit{New in version 3.7.}

\texttt{time.perf\_counter()} → float

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

\textit{New in version 3.3.}

\texttt{time.perf\_counter\_ns()} → int

Similar to \texttt{perf\_counter()}, but return time as nanoseconds.
New in version 3.7.

`time.process_time() → float`

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

New in version 3.3.

`time.process_time_ns() → int`

Similar to `process_time()` but return time as nanoseconds.

New in version 3.7.

`time.sleep(secs)`

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal’s catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

Changed in version 3.5: The function now sleeps at least `secs` even if the sleep is interrupted by a signal, except if the signal handler raises an exception (see PEP 475 for the rationale).

`time.strftime(format[, t])`

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the `format` argument. If `t` is not provided, the current time as returned
by `localtime()` is used. `format` must be a string. `ValueError` is raised if any field in `t` is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the `format` string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the `strftime()` result:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale’s abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale’s full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale’s abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale’s full month name.</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale’s appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>(1)</td>
</tr>
<tr>
<td>Directive</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>%%S</td>
<td>Second as a decimal number [00,61].</td>
<td></td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td></td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td></td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td></td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%z</td>
<td>Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].</td>
<td></td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name (no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal '%' character.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. When used with the `strftime()` function, the `%p` directive only affects the output hour field if the `%I` directive is used to
parse the hour.

2. The range really is 0 to 61; value 60 is valid in timestamps representing leap seconds and value 61 is supported for historical reasons.

3. When used with the `strptime()` function, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the RFC 2822 Internet email standard. [1]

```python
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime('Thu, 28 Jun 2001 14:17:15 +0000'))
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the `strftime(3)` documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial ` `%` of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

```python
time.strftime(string[, format])
```

Parse a string representing a time according to a format. The return value is a `struct_time` as returned by `gmtime()` or `localtime()`.

The `format` parameter uses the same directives as those used by `strftime()`; it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by `ctime()`. If `string` cannot be
parsed according to format, or if it has excess data after parsing, ValueError is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are \((1900, 1, 1, 0, 0, 0, 0, 1, -1)\). Both string and format must be strings.

For example:

```python
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y") # doctest: +NORMALIZE_WHITESPACE
(time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                    tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1))
```

Support for the %Z directive is based on the values contained in tzname and whether daylight is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because strftime() is implemented per platform it can sometimes offer more directives than those listed. But strptime() is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

**class time.struct_time**

The type of the time value sequence returned by gmtime(), localtime(), and strptime(). It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:
<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tm_year</td>
<td>(for example, 1993)</td>
</tr>
<tr>
<td>1</td>
<td>tm_mon</td>
<td>range [1, 12]</td>
</tr>
<tr>
<td>2</td>
<td>tm_mday</td>
<td>range [1, 31]</td>
</tr>
<tr>
<td>3</td>
<td>tm_hour</td>
<td>range [0, 23]</td>
</tr>
<tr>
<td>4</td>
<td>tm_min</td>
<td>range [0, 59]</td>
</tr>
<tr>
<td>5</td>
<td>tm_sec</td>
<td>range [0, 61]; see (2) in strftime() description</td>
</tr>
<tr>
<td>6</td>
<td>tm_wday</td>
<td>range [0, 6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td>tm_yday</td>
<td>range [1, 366]</td>
</tr>
<tr>
<td>8</td>
<td>tm_isdst</td>
<td>0, 1 or -1; see below</td>
</tr>
<tr>
<td>N/A</td>
<td>tm_zone</td>
<td>abbreviation of timezone name</td>
</tr>
<tr>
<td>N/A</td>
<td>tm_gmtoff</td>
<td>offset east of UTC in seconds</td>
</tr>
</tbody>
</table>

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

In calls to mktime(), tm_isdst may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a struct_time, or having elements of the wrong type, a TypeError is raised.

**time.time() → float**

Return the time in seconds since the epoch as a floating point number. The specific date of the epoch and the handling of leap
seconds is platform dependent. On Windows and most Unix systems, the epoch is January 1, 1970, 00:00:00 (UTC) and leap seconds are not counted towards the time in seconds since the epoch. This is commonly referred to as Unix time. To find out what the epoch is on a given platform, look at `gmtime(0)`.

Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

The number returned by `time()` may be converted into a more common time format (i.e. year, month, day, hour, etc...) in UTC by passing it to `gmtime()` function or in local time by passing it to the `localtime()` function. In both cases a `struct_time` object is returned, from which the components of the calendar date may be accessed as attributes.

```python
thread_time() -> float
```

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls in the same thread is valid.

**Availability:** Windows, Linux, Unix systems supporting `CLOCK_THREAD_CPUTIME_ID`.

*New in version 3.7.*

```python
thread_time_ns() -> int
```
Similar to `thread_time()` but return time as nanoseconds.

*New in version 3.7.*

```python
time.time() -> int
```

Similar to `time()` but returns time as an integer number of nanoseconds since the *epoch*.

*New in version 3.7.*

```python
time.tzset()
```

Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables `tzname` (from the TZ environment variable), `timezone` (non-DST seconds West of UTC), `altzone` (DST seconds west of UTC) and `daylight` (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

**Availability:** Unix.

**Note:** Although in many cases, changing the TZ environment variable may affect the output of functions like `localtime()` without calling `tzset()`, this behavior should not be relied on. The TZ environment variable should contain no whitespace.

The standard format of the TZ environment variable is (whitespace added for clarity):

```
std_offset [dst [offset [,start[/time], end[/time])]
```

Where the components are:
std and dst

Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

offset

The offset has the form: ± hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a ‘-‘, the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time]

Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn

The Julian day \( n \) (1 <= \( n \) <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n

The zero-based Julian day (0 <= \( n \) <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d

The \( d \)'th day (0 <= \( d \) <= 6) of week \( n \) of month \( m \) of the year (1 <= \( n \) <= 5, 1 <= \( m \) <= 12, where week 5 means “the last \( d \) day in month \( m \)” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the \( d \)'th day occurs. Day zero is a Sunday.

time has the same format as offset except that no leading sign (‘-‘ or ‘+‘) is allowed. The default, if time is not given, is 02:00:00.

```python
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
```
On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system’s zoneinfo (tzfile(5)) database to specify the timezone rules. To do this, set the TZ environment variable to the path of the required timezone datafile, relative to the root of the systems ‘zoneinfo’ timezone database, usually located at /usr/share/zoneinfo. For example, 'US/Eastern', 'Australia/Melbourne', 'Egypt' or 'Europe/Amsterdam'.

```python
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```
Clock ID Constants

These constants are used as parameters for `clock_getres()` and `clock_gettime()`.

**time. CLOCK_BOOTTIME**

Identical to `CLOCK_MONOTONIC`, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of `CLOCK_REALTIME`, which may have discontinuities if the time is changed using `settimeofday()` or similar.

*Availability*: Linux 2.6.39 or later.

*New in version 3.7.*

**time. CLOCK_HIGHRES**

The Solaris OS has a `CLOCK_HIGHRES` timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. `CLOCK_HIGHRES` is the nonadjustable, high-resolution clock.

*Availability*: Solaris.

*New in version 3.3.*

**time. CLOCK_MONOTONIC**

Clock that cannot be set and represents monotonic time since some unspecified starting point.

*Availability*: Unix.
New in version 3.3.

time. **CLOCK_MONOTONIC_RAW**
Similar to **CLOCK_MONOTONIC**, but provides access to a raw hardware-based time that is not subject to NTP adjustments.

**Availability**: Linux 2.6.28 and newer, macOS 10.12 and newer.

New in version 3.3.

time. **CLOCK_PROCESS_CPUTIME_ID**
High-resolution per-process timer from the CPU.

**Availability**: Unix.

New in version 3.3.

time. **CLOCK_PROF**
High-resolution per-process timer from the CPU.

**Availability**: FreeBSD, NetBSD 7 or later, OpenBSD.

New in version 3.7.

time. **CLOCK_THREAD_CPUTIME_ID**
Thread-specific CPU-time clock.

**Availability**: Unix.

New in version 3.3.

time. **CLOCK_UPTIME**
Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.
**Availability:** FreeBSD, OpenBSD 5.5 or later.

*New in version 3.7.*

The following constant is the only parameter that can be sent to `clock_settime()`.

**time.** `CLOCK_REALTIME`

System-wide real-time clock. Setting this clock requires appropriate privileges.

**Availability:** Unix.

*New in version 3.3.*
Timezone Constants

**time.altzone**

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if **daylight** is nonzero. See note below.

**time.daylight**

Nonzero if a DST timezone is defined. See note below.

**time.timezone**

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK). See note below.

**time.tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used. See note below.

**Note:** For the above Timezone constants (**altzone**, **daylight**, **timezone**, and **tzname**), the value is determined by the timezone rules in effect at module load time or the last time **tzset()** is called and may be incorrect for times in the past. It is recommended to use the **tm_gmtoff** and **tm_zone** results from **localtime()** to obtain timezone information.

**See also:**

Module **datetime**
More object-oriented interface to dates and times.

**Module locale**

Internationalization services. The locale setting affects the interpretation of many format specifiers in `strptime()` and `strftime()`.

**Module calendar**

General calendar-related functions. `timegm()` is the inverse of `gmtime()` from this module.

---

**Footnotes**

The use of `%Z` is now deprecated, but the `%z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, RFC 822 became obsolete and the 4-digit year has been first recommended by RFC 1123 and then mandated by RFC 2822.

[1]
argparse — Parser for command-line options, arguments and sub-commands

New in version 3.2.

Source code: Lib/argparse.py

The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv. The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.
Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```python
import argparse

parser = argparse.ArgumentParser(description='Process some integers.

parser.add_argument('integers', metavar='N', type=int, help='an integer for the accumulator

parser.add_argument('--sum', dest='accumulate', action='const', const=sum, default=max,
help='sum the integers (default: find the max)

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Assuming the Python code above is saved into a file called `prog.py`, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N          an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max
```

When run with the appropriate arguments, it prints either the sum or
the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4
$ python prog.py 1 2 3 4 --sum
10
```

If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

The following sections walk you through this example.

**Creating a parser**

The first step in using the `argparse` is creating an `ArgumentParser` object:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.'
```

The `ArgumentParser` object will hold all the information necessary to parse the command line into Python data types.

**Adding arguments**

Filling an `ArgumentParser` with information about program arguments is done by making calls to the `add_argument()` method. Generally, these calls tell the `ArgumentParser` how to take the strings on the command line and turn them into objects. This
information is stored and used when `parse_args()` is called. For example:

```python
>>> parser.add_argument('integers', metavar='N', type=int, help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', const=sum, default=max, help='sum the integers (default: find the max)')
```

Later, calling `parse_args()` will return an object with two attributes, `integers` and `accumulate`. The `integers` attribute will be a list of one or more ints, and the `accumulate` attribute will be either the `sum()` function, if `--sum` was specified at the command line, or the `max()` function if it was not.

### Parsing arguments

`ArgumentParser` parses arguments through the `parse_args()` method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple `Namespace` object will be built up from attributes parsed out of the command line:

```python
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, `parse_args()` will typically be called with no arguments, and the `ArgumentParser` will automatically determine the command-line arguments from `sys.argv`. 
ArgumentParser objects

class argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True)

Create a new ArgumentParser object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- **prog** - The name of the program (default: `sys.argv[0]`)
- **usage** - The string describing the program usage (default: generated from arguments added to parser)
- **description** - Text to display before the argument help (default: none)
- **epilog** - Text to display after the argument help (default: none)
- **parents** - A list of ArgumentParser objects whose arguments should also be included
- **formatter_class** - A class for customizing the help output
- **prefix_chars** - The set of characters that prefix optional arguments (default: `'-`)
- **fromfile_prefix_chars** - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- **argument_default** - The global default value for arguments (default: `None`)
- **conflict_handler** - The strategy for resolving conflicting optionals (usually unnecessary)
- **add_help** - Add a `-h/--help` option to the parser (default: `True`)
- **allow_abbrev** - Allows long options to be abbreviated if the
abbreviation is unambiguous. (default: \texttt{True})

*\textit{Changed in version 3.5: allow\_abbrev} parameter was added.*

The following sections describe how each of these are used.

**prog**

By default, \texttt{ArgumentParser} objects use \texttt{sys.argv[0]} to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named \texttt{myprogram.py} with the following code:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display \texttt{myprogram.py} as the program name (regardless of where the program was invoked from):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help   show this help message and exit
  --foo FOO    foo help
```

```
$ cd..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help   show this help message and exit
```
-foo FOO  foo help

To change this default behavior, another value can be supplied using the `prog=` argument to `ArgumentParser`:

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help       show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]` or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
  -h, --help       show this help message and exit
  --foo FOO        foo of the myprogram program
```

usage

By default, `ArgumentParser` calculates the usage message from the arguments it contains:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo')
>>> parser.add_argument('bar', nargs='+', help='bar')
```
The default message can be overridden with the `usage=` keyword argument:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%prog [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
    bar     bar help

optional arguments:
    -h, --help    show this help message and exit
    --foo [FOO]   foo help
```

The `%prog` format specifier is available to fill in the program name in your usage messages.

**description**

Most calls to the `ArgumentParser` constructor will use the `description=` keyword argument. This argument gives a brief
description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```python
>>> parser = argparse.ArgumentParser(description='A foo that bars',
>>> parser.print_help()
usage: argparse.py [-h]
A foo that bars
optional arguments:
  -h, --help  show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the `formatter_class` argument.

**epilog**

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to `ArgumentParser`:

```python
>>> parser = argparse.ArgumentParser(  
...     description='A foo that bars',  
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]
A foo that bars
optional arguments:
  -h, --help  show this help message and exit
```
And that's how you'd foo a bar

As with the description argument, the epilog= text is by default line-wrapped, but this behavior can be adjusted with the formatter_class argument to ArgumentParser.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to parents= argument to ArgumentParser can be used. The parents= argument takes a list of ArgumentParser objects, collects all the positional and optional actions from them, and adds these actions to the ArgumentParser object being constructed:

```python
>>> parent_parser = argparse.ArgumentParser(add_help=True, parents=argparse._null_parser)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify add_help=False. Otherwise, the ArgumentParser will see two -h/--help options (one in the parent and one in the child) and raise an error.
**Note:** You must fully initialize the parsers before passing them via parents=. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

**formatter_class**

`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

```python
class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
class argparse.MetavarTypeHelpFormatter
```

`RawDescriptionHelpFormatter` and `RawTextHelpFormatter` give more control over how textual descriptions are displayed. By default, `ArgumentParser` objects line-wrap the description and epilog texts in command-line help messages:

```python
>>> parser = argparse.ArgumentParser(
    ...     prog='PROG',
    ...     description='''this description was indented weird
    ...         but that is okay''',
    ...     epilog='''
    ...         likewise for this epilog whose white
    ...             be cleaned up and whose words will be wr
    ...                 across a couple lines'''

>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay
```
optional arguments:
- h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cl
will be wrapped across a couple lines

Passing  
RawDescriptionHelpFormatter  as
formatter_class= indicates that description and epilog are already
correctly formatted and should not be line-wrapped:

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''
...     Please do not mess up this text!
...     -----------------------------
...     I have indented it
...     exactly the way
...     I want it
...     '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
-----------------------------
    I have indented it
    exactly the way
    I want it

optional arguments:
- h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help
text, including argument descriptions. However, multiple new lines are
replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

**ArgumentDefaultsHelpFormatter** automatically adds information about default values to each of the argument help messages:

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter
>>> parser.add_argument('--foo', type=int, default=42)
>>> parser.add_argument('bar', nargs='*', default=[1])
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar	BAR! (default: [1, 2, 3])

optional arguments:
  -h, --help	show this help message and exit
  --foo FOO	FOO! (default: 42)
```

**MetavarTypeHelpFormatter** uses the name of the type argument for each argument as the display name for its values (rather than using the dest as the regular formatter does):

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  float
```
optional arguments:
  -h, --help  show this help message and exit
  --foo  int

prefix_chars

Most command-line options will use - as the prefix, e.g. -f/--foo. Parsers that need to support different or additional prefix characters, e.g. for options like +f or /foo, may specify them using the prefix_chars= argument to the ArgumentParser constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The prefix_chars= argument defaults to '-'. Supplying a set of characters that does not include - will cause -f/--foo options to be disallowed.

fromfile_prefix_chars

Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the fromfile_prefix_chars= argument is given to the ArgumentParser constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:
Arguments read from a file must by default be one per line (but see also convert_arg_line_to_args()) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression ['-f', 'foo', '@args.txt'] is considered equivalent to the expression ['-f', 'foo', '-f', 'bar'].

The fromfile_prefix_chars= argument defaults to None, meaning that arguments will never be treated as file references.

**argument_default**

Generally, argument defaults are specified either by passing a default to add_argument() or by calling the set_defaults() methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the argument_default= keyword argument to ArgumentParser. For example, to globally suppress attribute creation on parse_args() calls, we supply argument_default=SUPPRESS:

```python
>>> parser = argparse.ArgumentParser(argument_default=None)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args([('--foo', '1', 'BAR'])
```
allow_abbrev

Normally, when you pass an argument list to the `parse_args()` method of an `ArgumentParser`, it recognizes abbreviations of long options.

This feature can be disabled by setting `allow_abbrev` to `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

New in version 3.5.

conflict_handler

`ArgumentParser` objects do not allow two actions with the same option string. By default, `ArgumentParser` objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
```
Arguments:

```
ArgumentError: argument --foo: conflicting option string(s):
```

Sometimes (e.g. when using parents) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value 'resolve' can be supplied to the conflict_handler= argument of ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
>>> parser.add_argument('-f', '--foo', help='old foo help',
>>> parser.add_argument('--foo', help='new foo help',
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]
optional arguments:
   -h, --help   show this help message and exit
   -f FOO       old foo help
   --foo FOO    new foo help
```

Note that ArgumentParser objects only remove an action if all of its option strings are overridden. So, in the example above, the old -f/- -foo action is retained as the -f action, because only the --foo option string was overridden.

add_help

By default, ArgumentParser objects add an option which simply displays the parser's help message. For example, consider a file named myprogram.py containing the following code:

```
import argparse
parser = argparse.ArgumentParser()
```
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()

If `-h` or `--help` is supplied at the command line, the ArgumentParser help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo help
```

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help` argument to `ArgumentParser`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
  --foo FOO     foo help
```

The help option is typically `-h`/-`--help`. The exception to this is if the `prefix_chars` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+')
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [+h]
```
optional arguments:
  +h, ++help  show this help message and exit
The add_argument() method

ArgumentParser.add_argument(name or flags..., [action][, nargs..., [const][, default][, type][, choices][, required][, help][, metavar][, dest])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- **name or flags** - Either a name or a list of option strings, e.g. `foo` or `-f`, `--foo`.
- **action** - The basic type of action to be taken when this argument is encountered at the command line.
- **nargs** - The number of command-line arguments that should be consumed.
- **const** - A constant value required by some action and nargs selections.
- **default** - The value produced if the argument is absent from the command line.
- **type** - The type to which the command-line argument should be converted.
- **choices** - A container of the allowable values for the argument.
- **required** - Whether or not the command-line option may be omitted (optionals only).
- **help** - A brief description of what the argument does.
- **metavar** - A name for the argument in usage messages.
- **dest** - The name of the attribute to be added to the object returned by parse_args().

The following sections describe how each of these are used.

**name or flags**
The `add_argument()` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like:

```python
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```python
>>> parser.add_argument('bar')
```

When `parse_args()` is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

**action**

`ArgumentParser` objects associate command-line arguments with actions. These actions can do just about anything with the command-
line arguments associated with them, though most actions simply add an attribute to the object returned by `parse_args()`. The action keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- **'store'** - This just stores the argument's value. This is the default action. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- **'store_const'** - This stores the value specified by the `const` keyword argument. The `store_const` action is most commonly used with optional arguments that specify some sort of flag. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const')
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- **'store_true'** and **'store_false'** - These are special cases of `store_const` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.add_argument('--baz', action='store_false')
  >>> parser.parse_args('--foo --bar'.split())
  ```
• 'append' - This stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times. Example usage:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

• 'append_const' - This stores a list, and appends the value specified by the `const` keyword argument to the list. (Note that the `const` keyword argument defaults to `None`.) The 'append_const' action is typically useful when multiple arguments need to store constants to the same list. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const')
>>> parser.add_argument('--int', dest='types', action='append_const')
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

• 'count' - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```
- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See `ArgumentParser` for details of how the output is created.

- `'version'` - This expects a `version=` keyword argument in the `add_argument()` call, and prints version information and exits when invoked:

```python
>>> import argparse
g>>> parser = argparse.ArgumentParser(prog='PROG')
g>>> parser.add_argument('--version', action='version')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The recommended way to do this is to extend `Action`, overriding the `__call__` method and optionally the `__init__` method.

An example of a custom action:

```python
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, nargs, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
```
For more details, see `Action`.

**nargs**

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. The supported values are:

- **N** (an integer). `N` arguments from the command line will be gathered together into a list. For example:

  ```python
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

- **'?**'. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from `default` will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from `const` will be produced.
Some examples to illustrate this:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const=None, default=None)
>>> parser.add_argument('bar', nargs='?', default=None)
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

One of the more common uses of `nargs='?'` is to allow optional input and output files:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=..., default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=..., default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>, outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>, outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`. All command-line arguments present are gathered into a list. Note that it generally doesn’t make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
```
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2').
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])

- '+' Just like '*', all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present. For example:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required
```

- `argparse.REMAINDER`. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd')
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the `action`. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.
const

The const argument of `add_argument()` is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions. The two most common uses of it are:

- When `add_argument()` is called with `action='store_const'` or `action='append_const'`. These actions add the const value to one of the attributes of the object returned by `parse_args()`. See the action description for examples.
- When `add_argument()` is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of const will be assumed instead. See the nargs description for examples.

With the 'store_const' and 'append_const' actions, the const keyword argument must be given. For other actions, it defaults to None.

default

All optional arguments and some positional arguments may be omitted at the command line. The default keyword argument of `add_argument()`, whose value defaults to None, specifies what value should be used if the command-line argument is not present. For optional arguments, the default value is used when the option string was not present at the command line:
If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any `type` conversion argument, if provided, before setting the attribute on the `Namespace` return value. Otherwise, the parser uses the value as is:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='42')
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

For positional arguments with `nargs` equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default='42')
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:
By default, `ArgumentParser` objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a `float` or `int`. The `type` keyword argument of `add_argument()` allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the `type` argument:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args(['2 temp.txt'].split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

See the section on the `default` keyword argument for information on when the `type` argument is applied to default arguments.

To ease the use of various types of files, the argparse module provides the factory `FileType` which takes the `mode=`, `bufsize=`, `encoding=` and `errors=` arguments of the `open()` function. For example, `FileType('w')` can be used to create a writable file:

```python
>>> parser = argparse.ArgumentParser()
```
>>> parser.add_argument('bar', type=argparse.FileType)
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)

The `type` can take any callable that takes a single string argument and returns the converted value:

```python
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
... >>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

The `choices` keyword argument may be more convenient for type checkers that simply check against a range of values:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```
See the `choices` section for more details.

**choices**

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the `choices` keyword argument to `add_argument()`. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```python
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
```

```python
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire'
```

Note that inclusion in the `choices` container is checked after any `type` conversions have been performed, so the type of the objects in the `choices` container should match the `type` specified:

```python
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
```

```python
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any object that supports the `in` operator can be passed as the
choices value, so dict objects, set objects, custom containers, etc. are all supported.

required

In general, the argparse module assumes that flags like -f and --bar indicate optional arguments, which can always be omitted at the command line. To make an option required, True can be specified for the required= keyword argument to add_argument():

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

As the example shows, if an option is marked as required, parse_args() will report an error if that option is not present at the command line.

**Note:** Required options are generally considered bad form because users expect options to be optional, and thus they should be avoided when possible.

help

The help value is a string containing a brief description of the argument. When a user requests help (usually by using -h or --help at the command line), these help descriptions will be displayed with each argument:
The help strings can include various format specifiers to avoid repetition of things like the program name or the argument default. The available specifiers include the program name, %(prog)s, and most keyword arguments to add_argument(), e.g. %(default)s, %(type)s, etc.:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int,
...                      help='the bar to %(prog)s (default: 42)')
```

usage: frobble [-h] [bar]

positional arguments:
  bar         the bar to frobble (default: 42)

optional arguments:
  -h, --help  show this help message and exit
```
appear in the help string, you must escape it as `%%`.

`argparse` supports silencing the help entry for certain options, by setting the help value to `argparse.SUPPRESS`:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]
optional arguments:
  -h, --help    show this help message and exit
```

**metavar**

When `ArgumentParser` generates help messages, it needs some way to refer to each expected argument. By default, `ArgumentParser` objects use the `dest` value as the “name” of each object. By default, for positional argument actions, the `dest` value is used directly, and for optional argument actions, the `dest` value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar
positional arguments:
```
optional arguments:
-h, --help    show this help message and exit
--foo FOO

An alternative name can be specified with `metavar`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')

usage: [-h] [--foo YYY] XXX

positional arguments:
  XXX

optional arguments:
  -h, --help    show this help message and exit
  --foo YYY
```

Note that `metavar` only changes the *displayed* name - the name of
the attribute on the `parse_args()` object is still determined by the
`dest` value.

Different values of `nargs` may cause the metavar to be used multiple
times. Providing a tuple to `metavar` specifies a different display for
each of the arguments:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar',))
>>> parser.print_help()
```
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
    -h,  --help           show this help message and exit
    -x X X
    --foo bar baz

dest

Most ArgumentParser actions add some value as an attribute of the object returned by parse_args(). The name of this attribute is determined by the dest keyword argument of add_argument(). For positional argument actions, dest is normally supplied as the first argument to add_argument():

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of dest is normally inferred from the option strings. ArgumentParser generates the value of dest by taking the first long option string and stripping away the initial -- string. If no long option strings were supplied, dest will be derived from the first short option string by stripping the initial - character. Any internal - characters will be converted to _ characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
```
dest allows a custom attribute name to be provided:

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')

Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the action parameter to add_argument().

class argparse.Action(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to ArgumentParser.add_argument() except for the action itself.

Instances of Action (or return value of any callable to the action parameter) should have attributes “dest”, “option_strings”, “default”, “type”, “required”, “help”, etc. defined. The easiest way to ensure these attributes are defined is to call Action.__init__.
Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters:

- **parser** - The ArgumentParser object which contains this action.
- **namespace** - The Namespace object that will be returned by `parse_args()`. Most actions add an attribute to this object using `setattr()`.
- **values** - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the `type` keyword argument to `add_argument()`.
- **option_string** - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__` method may perform arbitrary actions, but will typically set attributes on the namespace based on `dest` and `values`.
The parse_args() method

ArgumentParser.parse_args(args=None, namespace=None)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to add_argument() determine exactly what objects are created and how they are assigned. See the documentation for add_argument() for details.

- **args** - List of strings to parse. The default is taken from sys.argv.
- **namespace** - An object to take the attributes. The default is a new empty Namespace object.

Option value syntax

The parse_args() method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:
For short options (options only one character long), the option and its value can be concatenated:

```python
>>> parser.parse_args([('--foo=FOO')])
Namespace(foo='FOO', x=None)
```

Several short options can be joined together, using only a single `-` prefix, as long as only the last option (or none of them) requires a value:

```python
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Invalid arguments

While parsing the command line, `parse_args()` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')
>>> # invalid type
>>> parser.parse_args([('--foo', 'spam')])
usage: PROG [-h] [--foo FOO] [bar]
```
Arguments containing `-`

The `parse_args()` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args()` method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')
```
If you have positional arguments that must begin with `-` and don’t look like negative numbers, you can insert the pseudo-argument `--` which tells `parse_args()` that everything after that is a positional argument:

```python
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

**Argument abbreviations (prefix matching)**

The `parse_args()` method **by default** allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
```

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')
```

```python
>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')
```

```python
>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2
```

```python
>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

```python
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```
An error is produced for arguments that could produce more than one options. This feature can be disabled by setting \texttt{allow\_abbrev} to \texttt{False}.

\section*{Beyond \texttt{sys.argv}}

Sometimes it may be useful to have an \texttt{ArgumentParser} parse arguments other than those of \texttt{sys.argv}. This can be accomplished by passing a list of strings to \texttt{parse\_args()}. This is useful for testing at the interactive prompt:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(...
...   'integers', metavar='int', type=int, choices ...
...   nargs='+', help='an integer in the range 0..',
>>> parser.add_argument(...
...   '--sum', dest='accumulate', action='store_const' ...
...   default=max, help='sum the integers (default: find the max',
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```
The Namespace object

`class argparse.Namespace`  
Simple class used by default by `parse_args()` to create an object holding attributes and return it.

This class is deliberately simple, just an `object` subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, `vars()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an `ArgumentParser` assign attributes to an already existing object, rather than a new `Namespace` object. This can be achieved by specifying the `namespace=` keyword argument:

```python
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```
Other utilities

Sub-commands

ArgumentParser.add_subparsers([title][, description][, prog][, parser_class][, action][, option_string][, dest][, required][, help][, metavar])

Many programs split up their functionality into a number of sub-commands, for example, the svn program can invoke sub-commands like svn checkout, svn update, and svn commit. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. ArgumentParser supports the creation of such sub-commands with the add_subparsers() method. The add_subparsers() method is normally called with no arguments and returns a special action object. This object has a single method, add_parser(), which takes a command name and any ArgumentParser constructor arguments, and returns an ArgumentParser object that can be modified as usual.

Description of parameters:

- title - title for the sub-parser group in help output; by default “subcommands” if description is provided, otherwise uses title for positional arguments
- description - description for the sub-parser group in help output, by default None
- prog - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
• **parser_class** - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
• **action** - the basic type of action to be taken when this argument is encountered at the command line
• **dest** - name of the attribute under which sub-command name will be stored; by default None and no value is stored
• **required** - Whether or not a subcommand must be provided, by default False.
• **help** - help for sub-parser group in help output, by default None
• **metavar** - string presenting available sub-commands in help; by default it is None and presents sub-commands in form {cmd1, cmd2, ..}

Some example usage:

```python
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true')
>>> subparsers = parser.add_subparsers(help='sub-command help')

>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')

>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')

>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```
Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```python
g>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
   {a,b}   sub-command help
       a   a help
       b   b help

optional arguments:
   -h, --help   show this help message and exit
   --foo   foo help

g>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
   bar   bar help

optional arguments:
   -h, --help   show this help message and exit
```

```python
g>>> parser.parse_args(['a'])
usage: PROG a [-h] bar

positional arguments:
   bar   bar help

optional arguments:
   -h, --help   show this help message and exit
```
The `add_subparsers()` method also supports `title` and `description` keyword arguments. When either is present, the subparser’s commands will appear in their own group in the help output. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands', description='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage: [-h] {foo,bar} ...
optional arguments:
  -h, --help  show this help message and exit
subcommands:
  valid subcommands
  {foo,bar}  additional help
```

Furthermore, `add_parser` supports an additional `aliases` argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:
One particularly effective way of handling sub-commands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example:

```python
# sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...

>>> def bar(args):
...     print('((%s))' % args.z)
...

# create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()

# create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)

# create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)

# parse the args and call whatever function was
```
This way, you can let `parse_args()` do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the `add_subparsers()` call will work:

```
parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(dest='subparser_name')
subparser1 = subparsers.add_parser('1')
subparser1.add_argument('-x')
subparser2 = subparsers.add_parser('2')
subparser2.add_argument('y')
parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

**FileType objects**

```python
class argparse.FileType(mode='r', bufsize=-1, encoding=None, errors=None)
```

The `FileType` factory creates objects that can be passed to the `type` argument of `ArgumentParser.add_argument()`. 
Arguments that have `FileType` objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the `open()` function for more details):

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType)
>>> parser.add_argument('out', type=argparse.FileType)
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.FileIO name='raw.dat' mode='wb'>)

`FileType` objects understand the pseudo-argument `'- '` and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects:

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType)
>>> parser.parse_args(['- '])
Namespace(infile=_<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)

*New in version 3.4:* The `encodings` and `errors` keyword arguments.

**Argument groups**

`ArgumentParser.add_argument_group(title=None, description=None)`

By default, `ArgumentParser` groups command-line arguments into “positional arguments” and “optional arguments” when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method:
The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular `ArgumentParser`. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts `title` and `description` arguments which can be used to customize this display:

```python
>>> parser = argparse.ArgumentParser(prog='PROG',
>>> group1 = parser.add_argument_group('group1',
>>>     title='group1 description',
>>>     description='foo	foo help'
>>> group2 = parser.add_argument_group('group2',
>>>     title='group2 description',
>>>     description='--bar BAR bar help'
>>> parser.print_help()
usage: PROG [--bar BAR] foo

        usage: PROG [--foo FOO] bar

        usage: PROG [--foo FOO] bar

        usage: PROG [--foo FOO] bar

        usage: PROG [--foo FOO] bar
```
Note that any arguments not in your user-defined groups will end up back in the usual “positional arguments” and “optional arguments” sections.

Mutual exclusion

ArgumentParser. **add_mutually_exclusive_group**(required=False)
Create a mutually exclusive group. argparse will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The **add_mutually_exclusive_group()** method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
```
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required

Note that currently mutually exclusive argument groups do not support the title and description arguments of add_argument_group().

Parser defaults

ArgumentParser.set_defaults(**kwargs)
Most of the time, the attributes of the object returned by parse_args() will be fully determined by inspecting the command-line arguments and the argument actions. set_defaults() allows some additional attributes that are determined without any inspection of the command line to be added:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```
Parser-level defaults can be particularly useful when working with multiple parsers. See the `add_subparsers()` method for an example of this type.

`ArgumentParser.get_default(dest)`

Get the default value for a namespace attribute, as set by either `add_argument()` or by `set_defaults()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Printing help

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

`ArgumentParser.print_usage(file=None)`

Print a brief description of how the `ArgumentParser` should be invoked on the command line. If `file` is `None`, `sys.stdout` is assumed.

`ArgumentParser.print_help(file=None)`

Print a help message, including the program usage and information about the arguments registered with the `ArgumentParser`. If `file` is `None`, `sys.stdout` is assumed.

There are also variants of these methods that simply return a string instead of printing it:
ArgumentParser. **format_usage()**
Return a string containing a brief description of how the ArgumentParser should be invoked on the command line.

ArgumentParser. **format_help()**
Return a string containing a help message, including the program usage and information about the arguments registered with the ArgumentParser.

Partial parsing

ArgumentParser. **parse_known_args**(args=None, namespace=None)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the `parse_known_args()` method can be useful. It works much like `parse_args()` except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true'
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

**Warning:** Prefix matching rules apply to `parse_known_args()`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments
Customizing file parsing

ArgumentParser. `convert_arg_line_to_args(arg_line)`
Arguments that are read from a file (see the `fromfile_prefix_chars` keyword argument to the ArgumentParser constructor) are read one argument per line. `convert_arg_line_to_args()` can be overridden for fancier reading.

This method takes a single argument `arg_line` which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this:

```python
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting methods

ArgumentParser. `exit(status=0, message=None)`
This method terminates the program, exiting with the specified `status` and, if given, it prints a `message` before that.

ArgumentParser. `error(message)`
This method prints a usage message including the `message` to the
standard error and terminates the program with a status code of 2.

Intermixed parsing

ArgumentParser.parse_intermixed_args(args=None, namespace=None)

ArgumentParser.parse_known_intermixed_args(args=None, namespace=None)

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The parse_intermixed_args() and parse_known_intermixed_args() methods support this parsing style.

These parsers do not support all the argparse features, and will raise exceptions if unsupported features are used. In particular, subparsers, argparse.REMAINDER, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between parse_known_args() and parse_intermixed_args(): the former returns ['2', '3'] as unparsed arguments, while the latter collects all the positionals into rest.

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3')
```
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])

parse_known_intermixed_args() returns a two item tuple containing the populated namespace and the list of remaining argument strings. parse_intermixed_args() raises an error if there are any remaining unparsed argument strings.

New in version 3.7.
Upgrading optparse code

Originally, the `argparse` module had attempted to maintain compatibility with `optparse`. However, `optparse` was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in `optparse` had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The `argparse` module improves on the standard library `optparse` module in a number of ways including:

- Handling positional arguments.
- Supporting sub-commands.
- Allowing alternative option prefixes like `+` and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom type and action.

A partial upgrade path from `optparse` to `argparse`:

- Replace all `optparse.OptionParser.add_option()` calls with `ArgumentParser.add_argument()` calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called `options`, now in the `argparse` context is called `args`.
- Replace `optparse.OptionParser.disable_interspersed_args()` by using `parse_intermixed_args()` instead of
parse_args).

- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for type keyword arguments with the corresponding type objects (e.g. `int`, `float`, `complex`, etc).
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the `OptionParser` constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`. 
getopt — C-style parser for command line options

Source code: Lib/getopt.py

Note: The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt() function. Users who are unfamiliar with the C getopt() function or who would like to write less code and get better help and error messages should consider using the argparse module instead.

This module helps scripts to parse the command line arguments in sys.argv. It supports the same conventions as the Unix getopt() function (including the special meanings of arguments of the form ‘-‘ and ‘--‘). Long options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:

getopt.getopt(args, shortopts, longopts=[])

Parses command line options and parameter list. args is the argument list to be parsed, without the leading reference to the running program. Typically, this means sys.argv[1:]. shortopts is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (':'); i.e., the same format that Unix getopt() uses.

Note: Unlike GNU getopt(), after a non-option argument, all further arguments are considered also non-options. This is similar
to the way non-GNU Unix systems work.

`longopts`, if specified, must be a list of strings with the names of the long options which should be supported. The leading `--` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`=\)`. Optional arguments are not supported. To accept only long options, `shortopts` should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if `longopts` is `[ 'foo', 'frob' ]`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so `GetoptError` will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of `args`). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `-x`) or two hyphens for long options (e.g., `--long-option`), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

```
gnu_getopt(args, shortopts, longopts=[])  
```

This function works like `getopt()`, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The `getopt()` function stops processing options as soon as a non-option argument is encountered.
If the first character of the option string is `'+', or if the environment variable POSIXLY_CORRECT is set, then option processing stops as soon as a non-option argument is encountered.

`exception` getopt.**GetoptError**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

`exception` getopt.**error**

Alias for **GetoptError**; for backward compatibility.

An example using only Unix style options:

```python
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')[0]  # ellides a remaining argument
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```python
>>> s = '--condition=foo --testing --output-file abc.def --extra=x'.split()
>>> s
['--condition=foo', '--testing', '--output-file', 'abc.def', '--extra=x']
>>> optlist, args = getopt.getopt(s, 'c:d:x:')[0]  # ellides a remaining argument
>>> optlist
[('--condition=foo', ''), ('--testing', ''), ('--output-file', 'abc.def'), ('--extra=x', '')]
>>> args
`x`
In a script, typical usage is something like this:

```python
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v")
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -o not recognized"
        usage()
        sys.exit(2)

    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"

    # ...
```
Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the `argparse` module:

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ...
```

See also:

**Module argparse**

Alternative command line option and argument parsing library.
logging — Logging facility for Python

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to see the tutorials (see the links on the right).

The basic classes defined by the module, together with their functions, are listed below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.
Logger Objects

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

The name is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That’s because in a module, `__name__` is the module’s name in the Python package namespace.

```python
class logging.Logger
    propagate
```

If this attribute evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers’ handlers - neither the level nor filters of the ancestor loggers in question are considered.

If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.
The constructor sets this attribute to True.

**Note:** If you attach a handler to a logger and one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to True. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

**setLevel**(level)
Sets the threshold for this logger to level. Logging messages which are less severe than level will be ignored; logging messages which have severity level or higher will be emitted by whichever handler or handlers service this logger, unless a handler’s level has been set to a higher severity level than level.

When a logger is created, the level is set to NOTSET (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level WARNING.

The term ‘delegation to the parent’ means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor’s level is treated as the effective level of the
logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

See Logging Levels for a list of levels.

*Changed in version 3.2:* The level parameter now accepts a string representation of the level such as ‘INFO’ as an alternative to the integer constants such as INFO. Note, however, that levels are internally stored as integers, and methods such as e.g. getEffectiveLevel() and isEnabledFor() will return/expect to be passed integers.

**isEnabledFor(lvl)**

Indicates if a message of severity lvl would be processed by this logger. This method checks first the module-level level set by logging.disable(lvl) and then the logger's effective level as determined by getEffectiveLevel().

**getEffectiveLevel()**

Indicates the effective level for this logger. If a value other than NOTSET has been set using setLevel(), it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than NOTSET is found, and that value is returned. The value returned is an integer, typically one of logging.DEBUG, logging.INFO etc.

**getChild(suffix)**

Returns a logger which is a descendant to this logger, as
determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string.

New in version 3.2.

def debug(msg, *args, **kwargs)
Logs a message with level DEBUG on this logger. The `msg` is the message format string, and the `args` are the arguments which are merged into `msg` using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in `kwargs` which are inspected: `exc_info`, `stack_info`, and `extra`.

If `exc_info` does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) or an exception instance is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is `stack_info`, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying `exc_info`: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception
handlers.

You can specify `stack_info` independently of `exc_info`, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last)`: which is used when displaying exception frames.

The third keyword argument is `extra` which can be used to pass a dictionary which is used to populate the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s ' Protocol problem: %s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset')
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by
the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the Formatter has been set up with a format string which expects ‘clientip’ and ‘user’ in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the extra dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized Formatters would be used with particular Handlers.

*New in version 3.2:* The stack_info parameter was added.

*Changed in version 3.5:* The exc_info parameter can now accept exception instances.

### info(msg, *args, **kwargs)
Logs a message with level INFO on this logger. The arguments are interpreted as for debug().

### warning(msg, *args, **kwargs)
Logs a message with level WARNING on this logger. The arguments are interpreted as for debug().
Note: There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

**error**(*msg, *args, **kwargs*)
Logs a message with level ERROR on this logger. The arguments are interpreted as for `debug()`.

**critical**(*msg, *args, **kwargs*)
Logs a message with level CRITICAL on this logger. The arguments are interpreted as for `debug()`.

**log**(*lvl, msg, *args, **kwargs*)
Logs a message with integer level `lvl` on this logger. The other arguments are interpreted as for `debug()`.

**exception**(*msg, *args, **kwargs*)
Logs a message with level ERROR on this logger. The arguments are interpreted as for `debug()`. Exception info is added to the logging message. This method should only be called from an exception handler.

**addFilter**(*filter*)
Adds the specified filter `filter` to this logger.

**removeFilter**(*filter*)
Removes the specified filter `filter` from this logger.

**filter**(*record*)
Applies this logger’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a
false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

**addHandler**(*hdlr*)

Adds the specified handler *hdlr* to this logger.

**removeHandler**(*hdlr*)

Removes the specified handler *hdlr* from this logger.

**findCaller**(*stack_info=False*)

Finds the caller’s source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless `stack_info` is `True`.

**handle**(*record*)

Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using `filter()`.

**makeRecord**(*name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None, sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized `LogRecord` instances.

**hasHandlers**()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else
False. The method stops searching up the hierarchy whenever a logger with the 'propagate' attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

New in version 3.2.

Changed in version 3.7: Loggers can now be pickled and unpickled.
Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
</tr>
<tr>
<td>NOTSET</td>
<td>0</td>
</tr>
</tbody>
</table>
Handler Objects

Handlers have the following attributes and methods. Note that Handler is never instantiated directly; this class acts as a base for more useful subclasses. However, the __init__() method in subclasses needs to call Handler.__init__().

```python
class logging.Handler
    __init__(level=NOTSET)
        Initializes the Handler instance by setting its level, setting the list of filters to the empty list and creating a lock (using createLock()) for serializing access to an I/O mechanism.

createLock()
    Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire()
    Acquires the thread lock created with createLock().

release()
    Releases the thread lock acquired with acquire().

setLevel(level)
    Sets the threshold for this handler to level. Logging messages which are less severe than level will be ignored. When a handler is created, the level is set to NOTSET (which causes all messages to be processed).

See Logging Levels for a list of levels.

Changed in version 3.2: The level parameter now accepts a
string representation of the level such as ‘INFO’ as an alternative to the integer constants such as INFO.

**setFormatter(fmt)**
Sets the Formatter for this handler to fmt.

**addFilter(filter)**
Adds the specified filter filter to this handler.

**removeFilter(filter)**
Removes the specified filter filter from this handler.

**filter(record)**
Applies this handler’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

**flush()**
Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

**close()**
Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when `shutdown()` is called. Subclasses should ensure that this gets called from overridden close() methods.

**handle(record)**
Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the
actual emission of the record with acquisition/release of the I/O thread lock.

**handleError** *(record)*

This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

**format** *(record)*

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

**emit** *(record)*

Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

For a list of handlers included as standard, see `logging.handlers`. 
Formatter Objects

Formatter objects have the following attributes and methods. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the default value of '%(message)s' is used, which just includes the message in the logging call. To have additional items of information in the formatted output (such as a timestamp), keep reading.

A Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - such as the default value mentioned above making use of the fact that the user’s message and arguments are pre-formatted into a LogRecord’s message attribute. This format string contains standard Python %-style mapping keys. See section printf-style String Formatting for more information on string formatting.

The useful mapping keys in a LogRecord are given in the section on LogRecord attributes.

class logging.Formatter(fmt=None, datefmt=None, style='%')

Returns a new instance of the Formatter class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no fmt is specified, '%(message)s' is used. If no datefmt is specified, a format is used which is described in the formatTime() documentation.

The style parameter can be one of ‘%’, ‘{‘ or ‘$’ and determines
how the format string will be merged with its data: using one of %-
formatting, \texttt{str.format()} or \texttt{string.Template}. See Using
particular formatting styles throughout your application for more
information on using {\texttt{-} and ${\texttt{-}}$-formatting for log messages.

\textit{Changed in version 3.2:} The \texttt{style} parameter was added.

\texttt{format(record)}

The record’s attribute dictionary is used as the operand to a
string formatting operation. Returns the resulting string. Before
formatting the dictionary, a couple of preparatory steps are
carried out. The \texttt{message} attribute of the record is computed
using \texttt{msg \% args}. If the formatting string contains
\texttt{'(asctime)'}\texttt{, formatTime()} is called to format the event
time. If there is exception information, it is formatted using
\texttt{formatException()} and appended to the message. Note
that the formatted exception information is cached in attribute
\texttt{exc_text}. This is useful because the exception information can
be pickled and sent across the wire, but you should be careful if
you have more than one \texttt{Formatter} subclass which
customizes the formatting of exception information. In this case,
you will have to clear the cached value after a formatter has
done its formatting, so that the next formatter to handle the
event doesn’t use the cached value but recalculates it afresh.

If stack information is available, it’s appended after the
exception information, using \texttt{formatStack()} to transform it if
necessary.

\texttt{formatTime(record, datefmt=None)}

This method should be called from \texttt{format()} by a formatter
which wants to make use of a formatted time. This method can
be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if `datefmt` (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the format `'%Y-%m-%d %H:%M:%S,uuu'` is used, where the `uuu` part is a millisecond value and the other letters are as per the `time.strftime()` documentation. An example time in this format is `2003-01-23 00:29:50,411`. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, `time localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as `time localtime()` or `time gmttime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

*Changed in version 3.3:* Previously, the default format was hard-coded as in this example: `2010-09-06 22:38:15,292` where the part before the comma is handled by a strptime format string (`'%Y-%m-%d %H:%M:%S'`), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s,%03d'` — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and
default_msec_format (for appending the millisecond value).

**FormatException**(exc_info)
Formats the specified exception information (a standard exception tuple as returned by sys.exc_info()) as a string. This default implementation just uses traceback.print_exception(). The resulting string is returned.

**formatStack**(stack_info)
Formats the specified stack information (a string as returned by traceback.print_stack(), but with the last newline removed) as a string. This default implementation just returns the input value.
Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with ‘A.B’ will allow events logged by loggers ‘A.B’, ‘A.B.C’, ‘A.B.C.D’, ‘A.B.D’ etc. but not ‘A.BB’, ‘B.A.B’ etc. If initialized with the empty string, all events are passed.

```python
class logging.Filter(name=")
    Returns an instance of the Filter class. If name is specified, it names a logger which, together with its children, will have its events allowed through the filter. If name is the empty string, allows every event.

    filter(record)
    Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.
```

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using debug(), info(), etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger’s filter setting, unless the filter has also been applied to those descendant loggers.

You don’t actually need to subclass Filter: you can pass any instance which has a filter method with the same semantics.

*Changed in version 3.2:* You don’t need to create specialized Filter
classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see Using Filters to impart contextual information).
LogRecord Objects

LogRecord instances are created automatically by the Logger every time something is logged, and can be created manually via makeLogRecord() (for example, from a pickled event received over the wire).

```python
class logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None, sinfo=None)
```
Contains all the information pertinent to the event being logged.

The primary information is passed in msg and args, which are combined using `msg % args` to create the message field of the record.

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong> – The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.</td>
</tr>
<tr>
<td><strong>level</strong> – The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to two attributes of the LogRecord: levelno for the numeric value andlevelname for the corresponding level name.</td>
</tr>
<tr>
<td><strong>pathname</strong> – The full pathname of the source file where the logging call was made.</td>
</tr>
<tr>
<td><strong>lineno</strong> – The line number in the source file where the logging call was made.</td>
</tr>
<tr>
<td><strong>msg</strong> – The event description message, possibly a format string with placeholders for variable data.</td>
</tr>
</tbody>
</table>
- **args** – Variable data to merge into the `msg` argument to obtain the event description.
- **exc_info** – An exception tuple with the current exception information, or `None` if no exception information is available.
- **func** – The name of the function or method from which the logging call was invoked.
- **sinfo** – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

**getMessage()**

Returns the message for this `LogRecord` instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, `str()` is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

*Changed in version 3.2:* The creation of a `LogRecord` has been made more configurable by providing a factory which is used to create the record. The factory can be set using `getLogRecordFactory()` and `setLogRecordFactory()` (see this for the factory's signature).

This functionality can be used to inject your own values into a `LogRecord` at creation time. You can use the following pattern:

```python
class record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

old_factory = logging.getLogRecordFactory()
def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record
```
logging.setLogRecordFactory(record_factory)

With this pattern, multiple factories could be chained, and as long as they don’t overwrite each other’s attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.
LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (str.format()), you can use {attrname} as the placeholder in the format string. If you are using $-formatting (string.Template), use the form ${attrname}. In both cases, of course, replace attrname with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of {msecs:03d} would format a millisecond value of 4 as 004. Refer to the str.format() documentation for full details on the options available to you.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
</table>
| args           | You shouldn’t need to format this yourself. | The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a
<table>
<thead>
<tr>
<th>Field</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>%(asctime)s</td>
<td>Human-readable time when the LogRecord was created. By default this is of the form ‘2003-07-08 16:49:45,896’ (the numbers after the comma are millisecond portion of the time).</td>
</tr>
<tr>
<td>created</td>
<td>%(created)f</td>
<td>Time when the LogRecord was created (as returned by \texttt{time.time()}).</td>
</tr>
<tr>
<td>exc_info</td>
<td>You shouldn’t need to format this yourself.</td>
<td>Exception tuple (à la \texttt{sys.exc_info}) or, if no exception has occurred, \texttt{None}.</td>
</tr>
<tr>
<td>filename</td>
<td>%(filename)s</td>
<td>Filename portion of pathname.</td>
</tr>
<tr>
<td>funcName</td>
<td>%(funcName)s</td>
<td>Name of function containing the logging call.</td>
</tr>
<tr>
<td>levelno</td>
<td>%(levelno)s</td>
<td>Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).</td>
</tr>
<tr>
<td>Field</td>
<td>Format String</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lineno</td>
<td>%(lineno)d</td>
<td>Source line number where the logging call was issued (if available).</td>
</tr>
<tr>
<td>message</td>
<td>%(message)s</td>
<td>The logged message, computed as msg % args. This is set when <code>Formatter.format()</code> is invoked.</td>
</tr>
<tr>
<td>module</td>
<td>%(module)s</td>
<td>Module (name portion of filename).</td>
</tr>
<tr>
<td>msecs</td>
<td>%(msecs)d</td>
<td>Millisecond portion of the time when the LogRecord was created.</td>
</tr>
<tr>
<td>msg</td>
<td>You shouldn't need to format this yourself.</td>
<td>The format string passed in the original logging call. Merged with args to produce message, or an arbitrary object (see Using arbitrary objects as messages).</td>
</tr>
<tr>
<td>name</td>
<td>%(name)s</td>
<td>Name of the logger used to log the call.</td>
</tr>
<tr>
<td>pathname</td>
<td>%(pathname)s</td>
<td>Full pathname of the source file where the logging call was issued (if available).</td>
</tr>
<tr>
<td>process</td>
<td>%(process)d</td>
<td>Process ID (if available).</td>
</tr>
<tr>
<td>processName</td>
<td>%(processName)s</td>
<td>Process name (if available).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time in milliseconds</td>
</tr>
<tr>
<td>Field</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>relativeCreated</td>
<td>%(relativeCreated)d</td>
<td>When the LogRecord was created, relative to the time the logging module was loaded.</td>
</tr>
<tr>
<td>stack_info</td>
<td>You shouldn't need to format this yourself.</td>
<td>Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.</td>
</tr>
<tr>
<td>thread</td>
<td>%(thread)d</td>
<td>Thread ID (if available).</td>
</tr>
<tr>
<td>threadName</td>
<td>%(threadName)s</td>
<td>Thread name (if available).</td>
</tr>
</tbody>
</table>

*Changed in version 3.1: processName was added.*
LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

```python
class logging.LoggerAdapter(logger, extra)
    Returns an instance of LoggerAdapter initialized with an underlying Logger instance and a dict-like object.

    process(msg, kwargs)
    Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as extra to the constructor and adds it to kwargs using key ‘extra’. The return value is a (msg, kwargs) tuple which has the (possibly modified) versions of the arguments passed in.
```

In addition to the above, LoggerAdapter supports the following methods of Logger: debug(), info(), warning(), error(), exception(), critical(), log(), isEnabledFor(), getEffectiveLevel(), setLevel() and hasHandlers(). These methods have the same signatures as their counterparts in Logger, so you can use the two types of instances interchangeably.

*Changed in version 3.2:* The isEnabledFor(), getEffectiveLevel(), setLevel() and hasHandlers() methods were added to LoggerAdapter. These methods delegate to the underlying logger.
Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the signal module, you may not be able to use logging from within such handlers. This is because lock implementations in the threading module are not always re-entrant, and so cannot be invoked from such signal handlers.
Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

logging.getLogger(name=None)

Return a logger with the specified name or, if name is None, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like ‘a’, ‘a.b’ or ‘a.b.c.d’. Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.getLoggerClass()

Return either the standard Logger class, or the last class passed to setLoggerClass(). This function may be called from within a new class definition, to ensure that installing a customized Logger class will not undo customizations already applied by other code. For example:

```python
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.getLogRecordFactory()

Return a callable which is used to create a LogRecord.

*New in version 3.2:* This function has been provided, along with setLogRecordFactory(), to allow developers more control over how the LogRecord representing a logging event is
constructed.

See `setLogRecordFactory()` for more information about the how the factory is called.

**logging.getLogger()**

Logs a message with level DEBUG on the root logger. The `msg` is the message format string, and the `args` are the arguments which are merged into `msg` using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in `kwargs` which are inspected: `exc_info` which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) or an exception instance is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

The second optional keyword argument is `stack_info`, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying `exc_info`: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify `stack_info` independently of `exc_info`, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:
Stack (most recent call last):

This mimics the Traceback (most recent call last): which is used when displaying exception frames.

The third optional keyword argument is extra which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```python
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset')
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in extra should not clash with the keys used by the logging system. (See the Formatter documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the Formatter has been set up with a format string which expects ‘clientip’ and ‘user’ in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case,
you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized *Formatters* would be used with particular *Handlers*.

*New in version 3.2:* The *stack_info* parameter was added.

```
logging.info(msg, *args, **kwargs)
```

Logs a message with level *INFO* on the root logger. The arguments are interpreted as for `debug()`.

```
logging.warning(msg, *args, **kwargs)
```

Logs a message with level *WARNING* on the root logger. The arguments are interpreted as for `debug()`.

*Note:* There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

```
logging.error(msg, *args, **kwargs)
```

Logs a message with level *ERROR* on the root logger. The arguments are interpreted as for `debug()`.

```
logging.critical(msg, *args, **kwargs)
```

Logs a message with level *CRITICAL* on the root logger. The arguments are interpreted as for `debug()`.
logging.\texttt{exception}(\textit{msg}, \*\textit{args}, **\textit{kwargs})

Logs a message with level ERROR on the root logger. The arguments are interpreted as for \texttt{debug()}. Exception info is added to the logging message. This function should only be called from an exception handler.

logging.\texttt{log}(\textit{level}, \textit{msg}, \*\textit{args}, **\textit{kwargs})

Logs a message with level \textit{level} on the root logger. The other arguments are interpreted as for \texttt{debug()}. 

\begin{quote}
\textbf{Note:} The above module-level convenience functions, which delegate to the root logger, call \texttt{basicConfig()} to ensure that at least one handler is available. Because of this, they should not be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger \textit{before} the threads are started. In earlier versions of Python, due to a thread safety shortcoming in \texttt{basicConfig()}, this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.
\end{quote}

logging.\texttt{disable}(\textit{lvl}=\texttt{CRITICAL})

Provides an overriding level \textit{lvl} for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity \textit{lvl} and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If \texttt{logging.disable(logging.NOTSET)} is called, it effectively
removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than CRITICAL (this is not recommended), you won’t be able to rely on the default value for the lvl parameter, but will have to explicitly supply a suitable value.

*Changed in version 3.7:* The lvl parameter was defaulted to level CRITICAL. See Issue #28524 for more information about this change.

**logging.addLevelName(lvl, levelName)**

Associates level lvl with text levelName in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a Formatter formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

**Note:** If you are thinking of defining your own levels, please see the section on Custom Levels.

**logging.getLevelName(lvl)**

Returns the textual representation of logging level lvl. If the level is one of the predefined levels CRITICAL, ERROR, WARNING, INFO or DEBUG then you get the corresponding string. If you have associated levels with names using addLevelName() then the name you have associated with lvl is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the
string ‘Level %s’ % lvl is returned.

**Note:** Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the %(levelname)s format specifier (see LogRecord attributes).

*Changed in version 3.4:* In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

**logging.makeLogRecord(attrdict)**

Creates and returns a new LogRecord instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled LogRecord attribute dictionary, sent over a socket, and reconstituting it as a LogRecord instance at the receiving end.

**logging.basicConfig(**kwargs**)**

Does basic configuration for the logging system by creating a StreamHandler with a default Formatter and adding it to the root logger. The functions debug(), info(), warning(), error() and critical() will call basicConfig() automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

**Note:** This function should be called from the main thread before other threads are started. In versions of Python prior to
2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.</td>
</tr>
<tr>
<td>filemode</td>
<td>If filename is specified, open the file in this mode. Defaults to 'a'.</td>
</tr>
<tr>
<td>format</td>
<td>Use the specified format string for the handler.</td>
</tr>
<tr>
<td>datefmt</td>
<td>Use the specified date/time format, as accepted by <code>time.strftime()</code>.</td>
</tr>
<tr>
<td>style</td>
<td>If format is specified, use this style for the format string. One of '%', '{' or '$' for printf-style, <code>str.format()</code> or <code>string.Template</code> respectively. Defaults to '%'.</td>
</tr>
<tr>
<td>level</td>
<td>Set the root logger level to the specified level.</td>
</tr>
<tr>
<td>stream</td>
<td>Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with filename - if both are present, a ValueError is raised.</td>
</tr>
<tr>
<td>handlers</td>
<td>If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don’t already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with filename or stream - if both are present, a ValueError is raised.</td>
</tr>
</tbody>
</table>
Changed in version 3.2: The style argument was added.

Changed in version 3.3: The handlers argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. handlers together with stream or filename, or stream together with filename).

**logging.shutdown()**

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

**logging.setLoggerClass(klass)**

Tells the logging system to use the class `klass` when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

**logging.setLogRecordFactory(factory)**

Set a callable which is used to create a `LogRecord`.

| Parameters: | factory – The factory callable to be used to instantiate a log record. |

New in version 3.2: This function has been provided, along with `getLogRecordFactory()`, to allow developers more control over how the `LogRecord` representing a logging event is constructed.
The factory has the following signature:

```python
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None, **kwargs)
```

<table>
<thead>
<tr>
<th>name:</th>
<th>The logger name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>level:</td>
<td>The logging level (numeric).</td>
</tr>
<tr>
<td>fn:</td>
<td>The full pathname of the file where the logging call was made.</td>
</tr>
<tr>
<td>lno:</td>
<td>The line number in the file where the logging call was made.</td>
</tr>
<tr>
<td>msg:</td>
<td>The logging message.</td>
</tr>
<tr>
<td>args:</td>
<td>The arguments for the logging message.</td>
</tr>
<tr>
<td>exc_info:</td>
<td>An exception tuple, or None.</td>
</tr>
<tr>
<td>func:</td>
<td>The name of the function or method which invoked the logging call.</td>
</tr>
<tr>
<td>sinfo:</td>
<td>A stack traceback such as is provided by <code>traceback.print_stack()</code>, showing the call hierarchy.</td>
</tr>
<tr>
<td>kwargs:</td>
<td>Additional keyword arguments.</td>
</tr>
</tbody>
</table>
Module-Level Attributes

logging. lastResort

A “handler of last resort” is available through this attribute. This is a StreamHandler writing to sys.stderr with a level of WARNING, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to sys.stderr. This replaces the earlier error message saying that “no handlers could be found for logger XYZ”. If you need the earlier behaviour for some reason, lastResort can be set to None.

New in version 3.2.
Integration with the warnings module

The `captureWarnings()` function can be used to integrate logging with the `warnings` module.

`logging.captureWarnings(capture)`

This function is used to turn the capture of warnings by logging on and off.

If `capture` is `True`, warnings issued by the `warnings` module will be redirected to the logging system. Specifically, a warning will be formatted using `warnings.formatwarning()` and the resulting string logged to a logger named 'py.warnings' with a severity of WARNING.

If `capture` is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

See also:

**Module `logging.config`**

Configuration API for the logging module.

**Module `logging.handlers`**

Useful handlers included with the logging module.

**PEP 282 - A Logging System**

The proposal which described this feature for inclusion in the Python standard library.

**Original Python logging package**

This is the original source for the `logging` package. The version
of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the logging package in the standard library.
This section describes the API for configuring the logging module.
Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

`logging.config.dictConfig(config)`

Takes the logging configuration from a dictionary. The contents of this dictionary are described in Configuration dictionary schema below.

If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The `logging.config` module has a callable attribute
dictConfigClass which is initially set to DictConfigurator. You can replace the value of dictConfigClass with a suitable implementation of your own.

dictConfig() calls dictConfigClass passing the specified dictionary, and then calls the configure() method on the returned object to put the configuration into effect:

```python
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of DictConfigurator could call DictConfigurator.__init__() in its own __init__(), then set up custom prefixes which would be usable in the subsequent configure() call. dictConfigClass would be bound to this new subclass, and then dictConfig() could be called exactly as in the default, uncustomized state.

*New in version 3.2.*

logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)

Reads the logging configuration from a configparser-format file. The format of the file should be as described in Configuration file format. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

- **fname** – A filename, or a file-like object, or an instance derived from RawConfigParser. If
Parameters:

- a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a `ConfigParser` is instantiated, and the configuration read by it from the object passed in `fname`. If that has a `readline()` method, it is assumed to be a file-like object and read using `read_file()`; otherwise, it is assumed to be a filename and passed to `read()`.

- `defaults` – Defaults to be passed to the `ConfigParser` can be specified in this argument.

- `disable_existing_loggers` – If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.

`Changed in version 3.4:` An instance of a subclass of `RawConfigParser` is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

`logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT, verify=None)`

Starts up a socket server on the specified port, and listens for new
configurations. If no port is specified, the module’s default DEFAULT_LOGGING_CONFIG_PORT is used. Logging configurations will be sent as a file suitable for processing by dictConfig() or fileConfig(). Returns a Thread instance on which you can call start() to start the server, and which you can join() when appropriate. To stop the server, call stopListening().

The verify argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the verify callable can perform signature verification and/or decryption. The verify callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or None to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using struct.pack('>L', n).

**Note:** Because portions of the configuration are passed through eval(), use of this function may open its users to a security risk. While the function only binds to a socket on localhost, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls listen(). Specifically, if the
A process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's `listen()` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the verify argument to `listen()` to prevent unrecognised configurations from being applied.

---

**Changed in version 3.4:** The verify argument was added.

---

**Note:** If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use `dictConfig()` for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

---

`logging.config.stopListening()` stops the listening server which was created with a call to `listen()`. This is typically called before calling `join()` on the return value from `listen()`. 
Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named ‘console’ and then say that the logger named ‘startup’ will send its messages to the ‘console’ handler. These objects aren’t limited to those provided by the logging module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as sys.stderr. The syntax for describing these objects and connections is defined in Object connections below.

Dictionary Schema Details

The dictionary passed to dictConfig() must contain the following keys:

- **version** - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a ‘configuring dict’ is mentioned, it will be checked for the special ‘()’ key to see if a custom instantiation is required. If so, the mechanism described in User-defined objects below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- **formatters** - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to
configure the corresponding `Formatter` instance.

The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a `Formatter` instance.

- **filters** - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.

- **handlers** - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.

  All other keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
    console:
        class : logging.StreamHandler
        formatter: brief
```
```python
level : INFO
filters: [allow_foo]
stream : ext://sys.stdout
file:
   class : logging.handlers.RotatingFileHandler
   formatter: precise
   filename: logconfig.log
   maxBytes: 1024
   backupCount: 3
```

The handler with id console is instantiated as a `logging.StreamHandler`, using `sys.stdout` as the underlying stream. The handler with id file is instantiated as a `logging.handlers.RotatingFileHandler` with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- **loggers** - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

The configuring dict is searched for the following keys:

- **level** (optional). The level of the logger.
- **propagate** (optional). The propagation setting of the logger.
- **filters** (optional). A list of ids of the filters for this logger.
- **handlers** (optional). A list of ids of the handlers for this logger.

The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- **root** - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the
propagate setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to False, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing `fileConfig()` API.

If the specified value is True, the configuration is processed as described in the section on Incremental Configuration.

- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig()`. If absent, this parameter defaults to True. This value is ignored if *incremental* is True.

### Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the *incremental* key of a configuration dict is present
and is True, the system will completely ignore any formatters and filters entries, and process only the level settings in the handlers entries, and the level and propagate settings in the loggers and root entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```yaml
formatters:
    brief:
        # configuration for formatter with id 'brief' goes here
    precise:
        # configuration for formatter with id 'precise'
```
handlers:
  h1: #This is an id
      # configuration of handler with id 'h1' goes here
      formatter: brief
  h2: #This is another id
      # configuration of handler with id 'h2' goes here
      formatter: precise
loggers:
  foo.bar.baz:
      # other configuration for logger 'foo.bar.baz'
  handlers: [h1, h2]

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. foo.bar.baz. The ids for Formatters and Filters can be any string value (such as brief, precise above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named foo.bar.baz should have two handlers attached to it, which are described by the handler ids h1 and h2. The formatter for h1 is that described by id brief, and the formatter for h2 is that described by id precise.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-
Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a ‘factory’ - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `()`.

Here’s a concrete example:

```yaml
formatters:
    brief:
        format: '%(message)s'
    default:
        format: '%(asctime)s %(levelname)-8s %(name)-15s
        datefmt: '%Y-%m-%d %H:%M:%S'
    custom:
        (): my.package.customFormatterFactory
        bar: baz
        spam: 99.9
        answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:
respectively, and as these dictionaries do not contain the special key
\texttt{()}, the instantiation is inferred from the context: as a result, standard \texttt{logging.Formatter} instances are created. The configuration sub-dictionary for the third formatter, with id \texttt{custom}, is:

\begin{verbatim}
{
  '()': 'my.package.customFormatterFactory',
  'bar': 'baz',
  'spam': 99.9,
  'answer': 42
}
\end{verbatim}

and this contains the special key \texttt{()}, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the \texttt{remaining} items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id \texttt{custom} will be assumed to be returned by the call:

\begin{verbatim}
my.package.customFormatterFactory(bar='baz', spam=99.9)
\end{verbatim}
The key '( )' has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The '( )' also serves as a mnemonic that the corresponding value is a callable.

**Access to external objects**

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string 'sys.stderr'. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string 'ext://sys.stderr' is provided as a value in the configuration, then the ext:// will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the prefix is recognised, the suffix is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

**Access to internal objects**
As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value 'DEBUG' for a level in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the handlers, filters and formatter entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the `logging` module. For example, consider `logging.handlers.MemoryHandler`, which takes a target argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given target just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an alternate handler, the configuration system would not know that the alternate referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```yaml
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler

  alternate: cfg://handlers.file
```

The literal string `cfg://handlers.file` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by
str.format. Thus, given the following snippet:

```python
handlers:
    email:
        class: logging.handlers.SMTPHandler
        mailhost: localhost
        fromaddr: my_app@domain.tld
        toaddrs:
            - support_team@domain.tld
            - dev_team@domain.tld
        subject: Houston, we have a problem.
```

in the configuration, the string `cfg://handlers` would resolve to the dict with key `handlers`, the string `cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team.domain.tld'` and the string `cfg://handlers.email.toaddrs[0]` would resolve to the value `'support_team@domain.tld'`. The subject value could be accessed using either `cfg://handlers.email.subject` or, equivalently, `cfg://handlers.email[subject]`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to
if that fails.

**Import resolution and custom importers**

Import resolution, by default, uses the built-in `__import__()` function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the importer attribute of the DictConfigurator or its superclass, the BaseConfigurator class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with `staticmethod()`. For example:

```python
from importlib import import_module
from logging.config import BaseConfigurator
BaseConfigurator.importer = staticmethod(import_module)
```

You don’t need to wrap with `staticmethod()` if you’re setting the import callable on a configurator instance.
Configuration file format

The configuration file format understood by `fileConfig()` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named log01 in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called hand01 in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called form01 in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

**Note:** The `fileConfig()` API is older than the `dictConfig()` API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure `Filter` objects, which provide for filtering of messages beyond simple integer levels, using `fileConfig()`. If you need to have instances of `Filter` in your logging configuration, you will need to use `dictConfig()`. Note that future enhancements to configuration functionality will be added to `dictConfig()`, so it’s worth considering transitioning to this newer API when it’s convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07
```
The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of DEBUG, INFO, WARNING, ERROR, CRITICAL or NOTSET. For the root logger only, NOTSET means that all messages will be logged. Level values are `eval()`uated in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```
The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are not propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by `eval()` in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean ‘log everything’.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when `eval()`uated in the context of the `logging` package’s namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers,
or to the examples below, to see how typical entries are constructed. If not provided, it defaults to {}.

The optional kwargs entry, when `eval()`uated in the context of the logging package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to {}.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=((('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')
```
Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The format entry is the overall format string, and the datefmt entry is the strftime()-compatible date/time format string. If empty, the package substitutes something which is almost equivalent to specifying the date format string `%Y-%m-%d %H:%M:%S`. This format also specifies milliseconds, which are appended to the result of
using the above format string, with a comma separator. An example time in this format is 2003-01-23 00:29:50,411.

The class entry is optional. It indicates the name of the formatter’s class (as a dotted module and class name.) This option is useful for instantiating a Formatter subclass. Subclasses of Formatter can present exception tracebacks in an expanded or condensed format.

**Note:** Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen()` documentation for more information.

**See also:**

**Module logging**  
API reference for the logging module.

**Module logging.handlers**  
Useful handlers included with the logging module.
The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.
StreamHandler

The StreamHandler class, located in the core logging package, sends logging output to streams such as sys.stdout, sys.stderr or any file-like object (or, more precisely, any object which supports write() and flush() methods).

class logging.StreamHandler(stream=None)

Returns a new instance of the StreamHandler class. If stream is specified, the instance will use it for logging output; otherwise, sys.stderr will be used.

emit(record)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a terminator. If exception information is present, it is formatted using traceback.print_exception() and appended to the stream.

flush()

Flushes the stream by calling its flush() method. Note that the close() method is inherited from Handler and so does no output, so an explicit flush() call may be needed at times.

setStream(stream)

Sets the instance’s stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

| Parameters: | stream – The stream that the handler should use. |
Returns: the old stream, if the stream was changed, or None if it wasn't.

*New in version 3.7.*

*Changed in version 3.2:* The `StreamHandler` class now has a `terminator` attribute, default value `'\n'`, which is used as the terminator when writing a formatted record to a stream. If you don’t want this newline termination, you can set the handler instance's `terminator` attribute to the empty string. In earlier versions, the terminator was hardcoded as `'\n'`.
FileHandler

The FileHandler class, located in the core logging package, sends logging output to a disk file. It inherits the output functionality from StreamHandler.

```python
class logging.FileHandler(filename, mode='a', encoding=None, delay=False)
```

Returns a new instance of the FileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

*Changed in version 3.6*: As well as string values, Path objects are also accepted for the filename argument.

**close()**

Closes the file.

**emit(record)**

Outputs the record to the file.
NullHandler

New in version 3.1.

The NullHandler class, located in the core logging package, does not do any formatting or output. It is essentially a ‘no-op’ handler for use by library developers.

class logging.NullHandler
    Returns a new instance of the NullHandler class.

    emit(record)
    This method does nothing.

    handle(record)
    This method does nothing.

    createLock()
    This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.

See Configuring Logging for a Library for more information on how to use NullHandler.
WatchedFileHandler

The WatchedFileHandler class, located in the logging.handlers module, is a FileHandler which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as newsyslog and logrotate which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, ST_INO is not supported under Windows; stat() always returns zero for this value.

class logging.handlers.WatchedFileHandler(filename, mode='a', encoding=None, delay=False)

    Returns a new instance of the WatchedFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

    Changed in version 3.6: As well as string values, Path objects are also accepted for the filename argument.
**reopenIf Needed()**
Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

*New in version 3.6.*

**emit(record)**
Outputs the record to the file, but first calls `reopenIf Needed()` to reopen the file if it has changed.
The **BaseRotatingHandler** class, located in the `logging.handlers` module, is the base class for the rotating file handlers, **RotatingFileHandler** and **TimedRotatingFileHandler**. You should not need to instantiate this class, but it has attributes and methods you may need to override.

```python
class logging.handlers.BaseRotatingHandler(filename, mode, encoding=None, delay=False)
```

The parameters are as for `FileHandler`. The attributes are:

**namer**

If this attribute is set to a callable, the `rotation_filename()` method delegates to this callable. The parameters passed to the callable are those passed to `rotation_filename()`.

**Note:** The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.

*New in version 3.3.*

**rotator**

If this attribute is set to a callable, the `rotate()` method delegates to this callable. The parameters passed to the callable are those passed to `rotate()`.

*New in version 3.3.*
**rotation_filename**(default_name)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the ‘namer’ attribute of the handler, if it’s callable, passing the default name to it. If the attribute isn’t callable (the default is `None`), the name is returned unchanged.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>default_name – The default name for the log file.</th>
</tr>
</thead>
</table>

*New in version 3.3.*

**rotate**(source, dest)

When rotating, rotate the current log.

The default implementation calls the ‘rotator’ attribute of the handler, if it’s callable, passing the source and dest arguments to it. If the attribute isn’t callable (the default is `None`), the source is simply renamed to the destination.

| Parameters | source – The source filename. This is normally the base filename, e.g. ‘test.log’.
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dest – The destination filename. This is normally what the source is rotated to, e.g. ‘test.log.1’.</td>
</tr>
</tbody>
</table>

*New in version 3.3.*

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of `RotatingFileHandler` and `TimedRotatingFileHandler`. If either the namer or rotator
callable raises an exception, this will be handled in the same way as any other exception during an `emit()` call, i.e. via the `handleError()` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see Using a rotator and namer to customize log rotation processing.
RotatingFileHandler

The RotatingFileHandler class, located in the logging.handlers module, supports rotation of disk log files.

class logging.handlers.RotatingFileHandler(filename, mode='a', maxBytes=0, backupCount=0, encoding=None, delay=False)

Returns a new instance of the RotatingFileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, 'a' is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

You can use the maxBytes and backupCount values to allow the file to rollover at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly maxBytes in length; but if either of maxBytes or backupCount is zero, rollover never occurs, so you generally want to set backupCount to at least 1, and have a non-zero maxBytes. When backupCount is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a backupCount of 5 and a base file name of app.log, you would get app.log, app.log.1, app.log.2, up to app.log.5. The file being written to is always app.log. When this file is filled, it is closed and renamed to app.log.1, and if files app.log.1, app.log.2, etc. exist, then they are renamed to app.log.2, app.log.3 etc. respectively.
Changed in version 3.6: As well as string values, Path objects are also accepted for the filename argument.

**doRollover()**

Does a rollover, as described above.

**emit(record)**

Outputs the record to the file, catering for rollover as described previously.
TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

class
logging.handlers.TimedRotatingFileHandler(filename, when='h', interval=1, backupCount=0, encoding=None, delay=False, utc=False, atTime=None)

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of `when` and `interval`.

You can use the `when` to specify the type of `interval`. The list of possible values is below. Note that they are not case sensitive.

<table>
<thead>
<tr>
<th>Value</th>
<th>Type of interval</th>
<th>If/how <code>atTime</code> is used</th>
</tr>
</thead>
<tbody>
<tr>
<td>'S'</td>
<td>Seconds</td>
<td>Ignored</td>
</tr>
<tr>
<td>'M'</td>
<td>Minutes</td>
<td>Ignored</td>
</tr>
<tr>
<td>'H'</td>
<td>Hours</td>
<td>Ignored</td>
</tr>
<tr>
<td>'D'</td>
<td>Days</td>
<td>Ignored</td>
</tr>
<tr>
<td>'W0'-'W6'</td>
<td>Weekday (0=Monday)</td>
<td>Used to compute initial rollover time</td>
</tr>
<tr>
<td>midnight</td>
<td>Roll over at midnight, if <code>atTime</code> not specified, else at time <code>atTime</code></td>
<td>Used to compute initial rollover time</td>
</tr>
</tbody>
</table>

When using weekday-based rotation, specify ‘W0’ for Monday, ‘W1’
for Tuesday, and so on up to ‘W6’ for Sunday. In this case, the value passed for \textit{interval} isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the \texttt{strftime} format \texttt{%Y-%m-%d\_%H-%M-%S} or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the \textit{utc} argument is true, times in UTC will be used; otherwise local time is used.

If \textit{backupCount} is nonzero, at most \textit{backupCount} files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If \textit{delay} is true, then file opening is deferred until the first call to \texttt{emit()}.

If \textit{atTime} is not \texttt{None}, it must be a \texttt{datetime.time} instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen “at midnight” or “on a particular weekday”. Note that in these cases, the \textit{atTime} value is effectively used to compute the \textit{initial} rollover, and subsequent rollovers would be calculated via the normal interval calculation.

\textbf{Note:} Calculation of the initial rollover time is done when the
handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of “every minute” is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, then you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

*Changed in version 3.4:* `atTime` parameter was added.

*Changed in version 3.6:* As well as string values, `Path` objects are also accepted for the `filename` argument.

**doRollover()**

Does a rollover, as described above.

**emit(record)**

Outputs the record to the file, catering for rollover as described above.
SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

```python
class logging.handlers.SocketHandler(host, port)
```

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by `host` and `port`.

*Changed in version 3.4:* If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

`close()`

Closes the socket.

`emit()`

Pickles the record’s attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

`handleError()`

Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

`makeSocket()`
This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

**makePickle**(record)
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

**send**(packet)
Send a pickled string `packet` to the socket. This function allows for partial sends which can happen when the network is busy.

**createSocket**()
Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
• **retryFactor** (multiplier, defaulting to 2.0).
• **retryMax** (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won’t even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).
DatagramHandler

The `DatagramHandler` class, located in the `logging.handlers` module, inherits from `SocketHandler` to support sending logging messages over UDP sockets.

```python
class logging.handlers.DatagramHandler(host, port)
```

Returns a new instance of the `DatagramHandler` class intended to communicate with a remote machine whose address is given by `host` and `port`.

*Changed in version 3.4:* If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

**emit()**

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

**makeSocket()**

The factory method of `SocketHandler` is here overridden to create a UDP socket (`socket.SOCK_DGRAM`).

**send(s)**

Send a pickled string to a socket.
SysLogHandler

The **SysLogHandler** class, located in the `logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

```python
class logging.handlers.SysLogHandler(address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM)
```

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by `address` in the form of a `(host, port)` tuple. If `address` is not specified, (`'localhost', 514`) is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example `'/dev/log'`. In this case, a Unix domain socket is used to send the message to the syslog. If `facility` is not specified, `LOG_USER` is used. The type of socket opened depends on the `socktype` argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`.

Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually `'/dev/log'` but on OS/X it's `'/var/run/syslog'`. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.
Changed in version 3.2: socktype was added.

**close()**
Closes the socket to the remote host.

**emit**(record)
The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

*Changed in version 3.2.1:* (See: bpo-12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification (**RFC 5424**). More recent versions of these daemons don’t expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

*Changed in version 3.3:* (See: bpo-12419.) In earlier versions, there was no facility for an “ident” or “tag” prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every
message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

**encodePriority**(*facility, priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic LOG\_ values are defined in `SysLogHandler` and mirror the values defined in the `sys/syslog.h` header file.

### Priorities

<table>
<thead>
<tr>
<th>Name (string)</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>LOG_ALERT</td>
</tr>
<tr>
<td>crit or critical</td>
<td>LOG_CRIT</td>
</tr>
<tr>
<td>debug</td>
<td>LOG_DEBUG</td>
</tr>
<tr>
<td>emerg or panic</td>
<td>LOG_EMERG</td>
</tr>
<tr>
<td>err or error</td>
<td>LOG_ERR</td>
</tr>
<tr>
<td>info</td>
<td>LOG_INFO</td>
</tr>
<tr>
<td>notice</td>
<td>LOG_NOTICE</td>
</tr>
<tr>
<td>warn or warning</td>
<td>LOG_WARNING</td>
</tr>
</tbody>
</table>

### Facilities

<table>
<thead>
<tr>
<th>Name (string)</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>LOG_AUTH</td>
</tr>
<tr>
<td>authpriv</td>
<td>LOG_AUTHPRIV</td>
</tr>
<tr>
<td>cron</td>
<td>LOG_CRON</td>
</tr>
<tr>
<td>Name</td>
<td>Priority</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>daemon</td>
<td>LOG_DAEMON</td>
</tr>
<tr>
<td>ftp</td>
<td>LOG_FTP</td>
</tr>
<tr>
<td>kern</td>
<td>LOG_KERN</td>
</tr>
<tr>
<td>lpr</td>
<td>LOG_LPR</td>
</tr>
<tr>
<td>mail</td>
<td>LOG_MAIL</td>
</tr>
<tr>
<td>news</td>
<td>LOG_NEWS</td>
</tr>
<tr>
<td>syslog</td>
<td>LOG_SYSLOG</td>
</tr>
<tr>
<td>user</td>
<td>LOG_USER</td>
</tr>
<tr>
<td>uucp</td>
<td>LOG_UUCP</td>
</tr>
<tr>
<td>local0</td>
<td>LOG_LOCAL0</td>
</tr>
<tr>
<td>local1</td>
<td>LOG_LOCAL1</td>
</tr>
<tr>
<td>local2</td>
<td>LOG_LOCAL2</td>
</tr>
<tr>
<td>local3</td>
<td>LOG_LOCAL3</td>
</tr>
<tr>
<td>local4</td>
<td>LOG_LOCAL4</td>
</tr>
<tr>
<td>local5</td>
<td>LOG_LOCAL5</td>
</tr>
<tr>
<td>local6</td>
<td>LOG_LOCAL6</td>
</tr>
<tr>
<td>local7</td>
<td>LOG_LOCAL7</td>
</tr>
</tbody>
</table>

**mapPriority**(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps **DEBUG**, **INFO**, **WARNING**, **ERROR** and **CRITICAL** to the equivalent syslog names, and all other level names to 'warning'.
NTEventLogHandler

The NTEventLogHandler class, located in the logging.handlers module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond’s Win32 extensions for Python installed.

class logging.handlers.NTEventLogHandler(appname, dllname=None, logtype='Application')

Returns a new instance of the NTEventLogHandler class. The appname is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The dllname should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, 'win32service.pyd' is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The logtype is one of 'Application', 'System' or 'Security', and defaults to 'Application'.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.
**emit**(record)
Determines the message ID, event category and event type, and then logs the message in the NT event log.

**getEventCategory**(record)
Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

**getEventType**(record)
Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler’s typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler’s typemap attribute.

**getMessageID**(record)
Returns the message ID for the record. If you are using your own messages, you could do this by having the msg passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`. 
SMTPHandler

The **SMTPHandler** class, located in the `logging.handlers` module, supports sending logging messages to an email address via SMTP.

```python
class logging.handlers.SMTPHandler(mailhost, fromaddr, toaddr, subject, credentials=None, secure=None, timeout=1.0)
```

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The `toaddr` should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the `mailhost` argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the `credentials` argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the `secure` argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls()` method.)

A timeout can be specified for communication with the SMTP server using the `timeout` argument.

*New in version 3.3:* The `timeout` argument was added.

**emit**(record)

Formats the record and sends it to the specified addressees.

**getSubject**(record)
If you want to specify a subject line which is record-dependent, override this method.
MemoryHandler

The `MemoryHandler` class, located in the `logging.handlers` module, supports buffering of logging records in memory, periodically flushing them to a `target` handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

`MemoryHandler` is a subclass of the more general `BufferingHandler`, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

```python
class logging.handlers.BufferingHandler(capacity)
    Initializes the handler with a buffer of the specified capacity.

    emit(record)
        Appends the record to the buffer. If `shouldFlush()` returns true, calls `flush()` to process the buffer.

    flush()
        You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

    shouldFlush(record)
        Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.
```

```python
class logging.handlers.MemoryHandler(capacity, flushLevel=ERROR, target=None, flushOnClose=True)
    Returns a new instance of the `MemoryHandler` class. The
```
instance is initialized with a buffer size of \textit{capacity}. If \textit{flushLevel} is not specified, \textit{ERROR} is used. If no \textit{target} is specified, the target will need to be set using \texttt{setTarget()} before this handler does anything useful. If \textit{flushOnClose} is specified as \texttt{False}, then the buffer is \textit{not} flushed when the handler is closed. If not specified or specified as \texttt{True}, the previous behaviour of flushing the buffer will occur when the handler is closed.

\textit{Changed in version 3.6:} The \textit{flushOnClose} parameter was added.

\textbf{close()}

Calls \texttt{flush()}, sets the target to \texttt{None} and clears the buffer.

\textbf{flush()}

For a \texttt{MemoryHandler}, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

\textbf{setTarget(\textit{target})}

Sets the target handler for this handler.

\textbf{shouldFlush(\textit{record})}

Checks for buffer full or a record at the \textit{flushLevel} or higher.
HTTPHandler

The HTTPHandler class, located in the logging.handlers module, supports sending logging messages to a Web server, using either GET or POST semantics.

```python
class logging.handlers.HTTPHandler(host, url, method='GET', secure=False, credentials=None, context=None)
```

Returns a new instance of the HTTPHandler class. The host can be of the form host:port, should you need to use a specific port number. If no method is specified, GET is used. If secure is true, a HTTPS connection will be used. The context parameter may be set to a ssl.SSLContext instance to configure the SSL settings used for the HTTPS connection. If credentials is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP ‘Authorization’ header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

*Changed in version 3.5:* The context parameter was added.

```python
mapLogRecord(record)
```

Provides a dictionary, based on record, which is to be URL-encoded and sent to the web server. The default implementation just returns record.__dict__. This method can be overridden if e.g. only a subset of LogRecord is to be sent to the web server, or if more specific customization of what’s sent to the server is required.

```python
emit(record)
```
Sends the record to the Web server as a URL-encoded dictionary. The `mapLogRecord()` method is used to convert the record to the dictionary to be sent.

**Note:** Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using `setFormatter()` to specify a `Formatter` for a `HTTPHandler` has no effect. Instead of calling `format()`, this handler calls `mapLogRecord()` and then `urllib.parse.urlencode()` to encode the dictionary in a form suitable for sending to a Web server.
QueueHandler

New in version 3.2.

The **QueueHandler** class, located in the `logging.handlers` module, supports sending logging messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

Along with the **QueueListener** class, **QueueHandler** can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

```python
class logging.handlers.QueueHandler(queue)
    Returns a new instance of the QueueHandler class. The instance is initialized with the queue to send messages to. The queue can be any queue-like object; it’s used as-is by the enqueue() method, which needs to know how to send messages to it.

    emit(record)
        Enqueues the result of preparing the LogRecord.

    prepare(record)
        Prepares a record for queuing. The object returned by this method is enqueued.

        The base implementation formats the record to merge the message, arguments, and exception information, if present. It also removes unpickleable items from the record in-place.
```
You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

**enqueue**(record)

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.
QueueListener

New in version 3.2.

The QueueListener class, located in the logging.handlers module, supports receiving logging messages from a queue, such as those implemented in the queue or multiprocessing modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While QueueListener is not itself a handler, it is documented here because it works hand-in-hand with QueueHandler.

Along with the QueueHandler class, QueueListener can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via SMTPHandler) are done on a separate thread.

class logging.handlers.QueueListener(queue, *handlers, respect_handler_level=False)

Returns a new instance of the QueueListener class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the dequeue() method, which needs to know how to get messages from it. If respect_handler_level is True, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.
Changed in version 3.5: The `respect_handler_levels` argument was added.

**dequeue**(*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

**prepare**(*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

**handle**(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare()`.

**start()**

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

**stop()**

Stops the listener.
This asks the thread to terminate, and then waits for it to do so. Note that if you don’t call this before your application exits, there may be some records still left on the queue, which won't be processed.

**enqueue_sentinel()**

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

*New in version 3.3.*

**See also:**

- **Module `logging`**
  API reference for the logging module.
- **Module `logging.config`**
  Configuration API for the logging module.
getpass — Portable password input

Source code: Lib/getpass.py

The `getpass` module provides two functions:

`getpass.getpass(prompt='Password:', stream=None)`

Prompt the user for a password without echoing. The user is prompted using the string `prompt`, which defaults to `'Password:'`. On Unix, the prompt is written to the file-like object `stream` using the replace error handler if needed. `stream` defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stdout` (this argument is ignored on Windows).

If echo free input is unavailable `getpass()` falls back to printing a warning message to `stream` and reading from `sys.stdin` and issuing a `GetPassWarning`.

**Note:** If you call getpass from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

`exception getpass.GetPassWarning`

A `UserWarning` subclass issued when password input may be echoed.

`getpass.getuser()`

Return the “login name” of the user.
This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the `pwd` module, otherwise, an exception is raised.

In general, this function should be preferred over `os.getlogin()`.
curses — Terminal handling for character-cell displays

The curses module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

**Note:** Whenever the documentation mentions a character it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a character string it can be specified as a Unicode string or a byte string.

**Note:** Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the nl_langinfo function. That means that you have to call `locale.setlocale()` in the application and encode Unicode strings using one of the system’s available encodings. This example uses the system’s default encoding:

```python
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```
Then use `code` as the encoding for `str.encode()` calls.

See also:

**Module** `curses.ascii`
Utilities for working with ASCII characters, regardless of your locale settings.

**Module** `curses.panel`
A panel stack extension that adds depth to curses windows.

**Module** `curses.textpad`
Editable text widget for curses supporting Emacs-like bindings.

**Curses Programming with Python**
Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The `Tools/demo/` directory in the Python source distribution contains some example programs using the curses bindings provided by this module.
Functions

The module `curses` defines the following exception:

*exception* `curses.error`

Exception raised when a curses library function returns an error.

**Note:** Whenever `x` or `y` arguments to a function or a method are optional, they default to the current cursor location. Whenever `attr` is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions:

`curses.baudrate()`

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

`curses.beep()`

Emit a short attention sound.

`curses.can_change_color()`

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

`curses.cbreak()`

Enter cbreak mode. In cbreak mode (sometimes called “rare” mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode,
special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first `raw()` then `cbreak()` leaves the terminal in cbreak mode.

**curses.color_content(color_number)**
Return the intensity of the red, green, and blue (RGB) components in the color `color_number`, which must be between 0 and COLORS. Return a 3-tuple, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

**curses.color_pair(color_number)**
Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. `pair_number()` is the counterpart to this function.

**curses.curs_set(visibility)**
Set the cursor state. `visibility` can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

**curses.def_prog_mode()**
Save the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to `reset_prog_mode()` will restore this mode.

**curses.def_shell_mode()**
Save the current terminal mode as the “shell” mode, the mode
when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to \texttt{reset\_shell\_mode()} will restore this mode.

\texttt{curses.\texttt{delay\_output}(ms)}  
Insert an \texttt{ms} millisecond pause in output.

\texttt{curses.\texttt{doupdate}()}  
Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The \texttt{doupdate()} ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a \texttt{noutrefresh()} call after write operations such as \texttt{addstr()} have been performed on a window. The normal \texttt{refresh()} call is simply \texttt{noutrefresh()} followed by \texttt{doupdate()}; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing \texttt{noutrefresh()} calls on all windows, followed by a single \texttt{doupdate()}.

\texttt{curses.\texttt{echo}()}  
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

\texttt{curses.\texttt{endwin}()}  
De-initialize the library, and return terminal to normal status.

\texttt{curses.\texttt{erasechar}()}  
Return the user’s current erase character as a one-byte bytes
object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses\texttt{.filter()}

The \texttt{filter()} routine, if used, must be called before \texttt{initscr()} is called. The effect is that, during those calls, LINES is set to 1; the capabilities \texttt{clear}, \texttt{cup}, \texttt{cud}, \texttt{cud1}, \texttt{cuu1}, \texttt{cuu}, \texttt{vpa} are disabled; and the \texttt{home} string is set to the value of \texttt{cr}. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses\texttt{.flash()}

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as ‘visible bell’ to the audible attention signal produced by \texttt{beep()}.  

curses\texttt{.flushinp()}

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses\texttt{.getmouse()}

After \texttt{getch()} returns \texttt{KEY_MOUSE} to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple \texttt{(id, x, y, z, bstate)}. \texttt{id} is an ID value used to distinguish multiple devices, and \texttt{x, y, z} are the event’s coordinates. (\texttt{z} is currently unused.) \texttt{bstate} is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where \texttt{n} is the button number from 1 to 4: \texttt{BUTTONn_PRESSED},
BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

curses.\texttt{getsyx()}
Return the current coordinates of the virtual screen cursor as a tuple $ (y, x) $. If \texttt{leaveok} is currently True, then return $ (-1, -1) $.

curses.\texttt{getwin(file)}
Read window related data stored in the file by an earlier \texttt{putwin()} call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.\texttt{has\_colors()}
Return True if the terminal can display colors; otherwise, return False.

curses.\texttt{has\_ic()}
Return True if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.\texttt{has\_il()}
Return True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.\texttt{has\_key(ch)}
Take a key value \texttt{ch}, and return True if the current terminal type recognizes a key with that value.
curses.\texttt{halfdelay}(tenths)

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for \textit{tenths} tenths of seconds, raise an exception if nothing has been typed. The value of \textit{tenths} must be a number between 1 and 255. Use \texttt{nocbreak()} to leave half-delay mode.

curses.\texttt{init\_color}(color\_number, r, g, b)

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of \textit{color\_number} must be between 0 and COLORS. Each of \textit{r}, \textit{g}, \textit{b}, must be a value between 0 and 1000. When \texttt{init\_color()} is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if \texttt{can\_change\_color()} returns True.

curses.\texttt{init\_pair}(pair\_number, fg, bg)

Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of \textit{pair\_number} must be between 1 and COLOR\_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of \textit{fg} and \textit{bg} arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.\texttt{initscr}()

Initialize the library. Return a \texttt{window} object which represents the whole screen.
Note: If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

curses.is_term_resized(nlines, ncols)

Return True if resize_term() would modify the window structure, False otherwise.

curses.isendwin()

Return True if endwin() has been called (that is, the curses library has been deinitialized).

curses.keyname(k)

Return the name of the key numbered k as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (b'^') followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix b'M-' followed by the name of the corresponding ASCII character.

curses.killchar()

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.longname()

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to initscr().
curses.meta(flag)

If flag is True, allow 8-bit characters to be input. If flag is False, allow only 7-bit chars.

curses.mouseinterval(interval)

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

curses.mousemask(mousemask)

Set the mouse events to be reported, and return a tuple (availmask, oldmask). availmask indicates which of the specified mouse events can be reported; on complete failure it returns 0. oldmask is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

curses.napms(ms)

Sleep for ms milliseconds.

curses.newpad(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The refresh() and noutrefresh()
methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol; the p arguments refer to the upper left corner of the pad region to be displayed and the s arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.newwin(nlines, ncols)
curses.newwin(nlines, ncols, begin_y, begin_x)

Return a new window, whose left-upper corner is at (begin_y, begin_x), and whose height/width is nlines/ncols.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.nl()

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.nocbreak()

Leave cbreak mode. Return to normal “cooked” mode with line buffering.

curses.noecho()

Leave echo mode. Echoing of input characters is turned off.

curses.nonl()

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of addch(‘\n’), which always does the equivalent of return and
line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

**curses.noqiflush()**
When the noqiflush() routine is used, normal flush of input and output queues associated with the INTR, QUIT, and SUSP characters will not be done. You may want to call noqiflush() in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

**curses.noraw()**
Leave raw mode. Return to normal “cooked” mode with line buffering.

**curses.pair_content(pair_number)**
Return a tuple (fg, bg) containing the colors for the requested color pair. The value of `pair_number` must be between 1 and COLOR_PAIRS - 1.

**curses.pair_number(attr)**
Return the number of the color-pair set by the attribute value `attr`. `color_pair()` is the counterpart to this function.

**curses.putp(str)**
Equivalent to tputs(str, 1, putchar); emit the value of a specified terminfo capability for the current terminal. Note that the output of `putp()` always goes to standard output.

**curses.qiflush([flag])**
If `flag` is False, the effect is the same as calling noqiflush(). If `flag` is True, or no argument is provided, the queues will be
flushed when these control characters are read.

curses.raw()

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.reset_prog_mode()

Restore the terminal to “program” mode, as previously saved by def_prog_mode().

curses.reset_shell_mode()

Restore the terminal to “shell” mode, as previously saved by def_shell_mode().

curses.resetty()

Restore the state of the terminal modes to what it was at the last call to savetty().

curses.resize_term(nlines, ncols)

Backend function used by resizeterm(), performing most of the work; when resizing the windows, resize_term() blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The resize_term() function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.resizeterm(nlines, ncols)

Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the
SIGWINCH handler).

curses.\texttt{savetty()}
Save the current state of the terminal modes in a buffer, usable by \texttt{resetty()}.  

curses.\texttt{setsyx(y, x)}
Set the virtual screen cursor to $y, x$. If $y$ and $x$ are both -1, then \texttt{leaveok} is set \texttt{True}.  

curses.\texttt{setupterm(term=None, fd=-1)}
Initialize the terminal. \texttt{term} is a string giving the terminal name, or \texttt{None}; if omitted or \texttt{None}, the value of the \texttt{TERM} environment variable will be used. \texttt{fd} is the file descriptor to which any initialization sequences will be sent; if not supplied or -1, the file descriptor for \texttt{sys.stdout} will be used.  

curses.\texttt{start\_color()}
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after \texttt{initscr()}.  

\texttt{start\_color()} initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the \texttt{curses} module, \texttt{COLORS} and \texttt{COLOR\_PAIRS}, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.  

curses.\texttt{termattrs()}
Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs
complete control over the appearance of the screen.

curses.**termname**()

Return the value of the environment variable TERM, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(capname)

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value -1 if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses.**tigetnum**(capname)

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value -2 if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses.**tigetstr**(capname)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return None if *capname* is not a terminfo “string capability”, or is canceled or absent from the terminal description.

curses.**tparm**(str[, ...])

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in b'\033[6;4H', the exact result depending on terminal type.

curses.**typeahead**(fd)

Specify that the file descriptor *fd* be used for typeahead checking. If
\textit{fd} is \texttt{-1}, then no typeahead checking is done.

The \texttt{curses} library does “line-breakout optimization” by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or douupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.\texttt{uncrtl}(\texttt{ch})

Return a bytes object which is a printable representation of the character \texttt{ch}. Control characters are represented as a caret followed by the character, for example as \texttt{b'\^C'}. Printing characters are left as they are.

curses.\texttt{ungetch}(\texttt{ch})

Push \texttt{ch} so the next \texttt{getch()} will return it.

\textbf{Note:} Only one \texttt{ch} can be pushed before \texttt{getch()} is called.

curses.\texttt{update_lines_cols}()

Update LINES and COLS. Useful for detecting manual screen resize.

\textit{New in version 3.5.}

curses.\texttt{unget_wch}(\texttt{ch})

Push \texttt{ch} so the next \texttt{get_wch()} will return it.

\textbf{Note:} Only one \texttt{ch} can be pushed before \texttt{get_wch()} is called.

\textit{New in version 3.3.}
curses.ungetmouse\((id, x, y, z, bstate)\)
    Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

curses.use_env\((flag)\)
    If used, this function should be called before initscr() or newterm are called. When flag is False, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

curses.use_default_colors()
    Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, init_pair(x, curses.COLOR_RED, -1) initializes, for instance, color pair x to a red foreground color on the default background.

curses.wrapper\(func,...\)
    Initialize curses and call another callable object, func, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object func is then passed the main window ‘stdscr’ as its first argument, followed by any other arguments passed to wrapper(). Before calling func, wrapper() turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and
disables the terminal keypad.
Window Objects

Window objects, as returned by \texttt{initscr()} and \texttt{newwin()} above, have the following methods and attributes:

\begin{verbatim}
window.addch(ch[, attr])
window.addch(y, x, ch[, attr])
\end{verbatim}

Paint character \texttt{ch} at \((y, x)\) with attributes \texttt{attr}, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

\textbf{Note}: Writing outside the window, subwindow, or pad raises a \texttt{curses.error}. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

\begin{verbatim}
window.addnstr(str, n[, attr])
window.addnstr(y, x, str, n[, attr])
\end{verbatim}

Paint at most \texttt{n} characters of the character string \texttt{str} at \((y, x)\) with attributes \texttt{attr}, overwriting anything previously on the display.

\begin{verbatim}
window.addstr(str[, attr])
window.addstr(y, x, str[, attr])
\end{verbatim}

Paint the character string \texttt{str} at \((y, x)\) with attributes \texttt{attr}, overwriting anything previously on the display.

\textbf{Note}: Writing outside the window, subwindow, or pad raises a \texttt{curses.error}. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised
after the string is printed.

```c
window. attroff(attr)
```
Remove attribute `attr` from the “background” set applied to all writes to the current window.

```c
window. attron(attr)
```
Add attribute `attr` from the “background” set applied to all writes to the current window.

```c
window. attrset(attr)
```
Set the “background” set of attributes to `attr`. This set is initially 0 (no attributes).

```c
window. bkgd(ch[, attr])
```
Set the background property of the window to the character `ch`, with attributes `attr`. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

```c
window. bkgdset(ch[, attr])
```
Set the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete.
line/character operations.

```plaintext
window. border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]])
```

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

**Note:** A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>Left side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>rs</td>
<td>Right side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>ts</td>
<td>Top</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>bs</td>
<td>Bottom</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>tl</td>
<td>Upper-left corner</td>
<td>ACS_ULCORNER</td>
</tr>
<tr>
<td>tr</td>
<td>Upper-right corner</td>
<td>ACS_URCORNER</td>
</tr>
<tr>
<td>bl</td>
<td>Bottom-left corner</td>
<td>ACS_LLCORNER</td>
</tr>
<tr>
<td>br</td>
<td>Bottom-right corner</td>
<td>ACS_LRCORNER</td>
</tr>
</tbody>
</table>

```plaintext
window. box([vertch, horch])
```

Similar to `border()`, but both `ls` and `rs` are `vertch` and both `ts` and `bs` are `horch`. The default corner characters are always used by this function.

```plaintext
window. chgat(attr)
window. chgat(num, attr)
window. chgat(y, x, attr)
```
window.chgat(y, x, num, attr)
Set the attributes of num characters at the current cursor position, or at position (y, x) if supplied. If num is not given or is -1, the attribute will be set on all the characters to the end of the line. This function moves cursor to position (y, x) if supplied. The changed line will be touched using the touchline() method so that the contents will be redisplayed by the next window refresh.

window.clear()
Like erase(), but also cause the whole window to be repainted upon next call to refresh().

window.clearok(flag)
If flag is True, the next call to refresh() will clear the window completely.

window.clrtobot()
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of clrtoeol() is performed.

window.clrtoeol()
Erase from cursor to the end of the line.

window.cursyncup()
Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.delch([y, x])
Delete any character at (y, x).

window.deleteln()
Delete the line under the cursor. All following lines are moved up by one line.

```python
window.derwin(begin_y, begin_x)
```

```python
window.derwin(nlines, ncols, begin_y, begin_x)
```

An abbreviation for “derive window”, `derwin()` is the same as calling `subwin()`, except that `begin_y` and `begin_x` are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

```python
window.echochar(ch, [attr])
```

Add character `ch` with attribute `attr`, and immediately call `refresh()` on the window.

```python
window.enclose(y, x)
```

Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

```python
window.encoding
```

Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with `window.subwin()`. By default, the locale encoding is used (see `locale.getpreferredencoding()`).

*New in version 3.3.*

```python
window.erase()
```

Clear the window.
window.\texttt{getbegyx()}

Return a tuple \((y, x)\) of co-ordinates of upper-left corner.

window.\texttt{getbkgd()}

Return the given window's current background character/attribute pair.

window.\texttt{getch([y, x])}

Get a character. Note that the integer returned does \textit{not} have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return \texttt{-1} if there is no input, otherwise wait until a key is pressed.

window.\texttt{get_wch([y, x])}

Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

\textit{New in version 3.3.}

window.\texttt{getkey([y, x])}

Get a character, returning a string instead of an integer, as \texttt{getch()} does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.\texttt{getmaxyx()}

Return a tuple \((y, x)\) of the height and width of the window.

window.\texttt{getparyx()}

Return the beginning coordinates of this window relative to its parent window as a tuple \((y, x)\). Return \((-1, -1)\) if this
window has no parent.

window.getstr()
window.getstr(n)
window.getstr(y, x)
window.getstr(y, x, n)

Read a bytes object from the user, with primitive line editing capacity.

window.getyx()

Return a tuple \((y, x)\) of current cursor position relative to the window's upper-left corner.

window.hline(ch, n)
window.hline(y, x, ch, n)

Display a horizontal line starting at \((y, x)\) with length \(n\) consisting of the character \(ch\).

window.idcok(flag)

If \(flag\) is False, curses no longer considers using the hardware insert/delete character feature of the terminal; if \(flag\) is True, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.idlok(flag)

If \(flag\) is True, curses will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.immedok(flag)

If \(flag\) is True, any change in the window image automatically causes the window to be refreshed; you no longer have to call refresh() yourself. However, it may degrade performance
considerably, due to repeated calls to wrefresh. This option is disabled by default.

`window.inch([y, x])`

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

`window.insch(ch[, attr])`
`window.insch(y, x, ch[, attr])`

Paint character `ch` at `(y, x)` with attributes `attr`, moving the line from position `x` right by one character.

`window.insdelln(nlines)`

Insert `nlines` lines into the specified window above the current line. The `nlines` bottom lines are lost. For negative `nlines`, delete `nlines` lines starting with the one under the cursor, and move the remaining lines up. The bottom `nlines` lines are cleared. The current cursor position remains the same.

`window.insertln()`

Insert a blank line under the cursor. All following lines are moved down by one line.

`window.insnstr(str, n[, attr])`
`window.insnstr(y, x, str, n[, attr])`

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to `n` characters. If `n` is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).
window.insstr(str[, attr])
window.insstr(y, x, str[, attr])

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to y, x, if specified).

window.instr([n])
window.instr(y, x[, n])

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at y, x if specified. Attributes are stripped from the characters. If n is specified, instr() returns a string at most n characters long (exclusive of the trailing NUL).

window.is_linetouched(line)

Return True if the specified line was modified since the last call to refresh(); otherwise return False. Raise a curses.error exception if line is not valid for the given window.

window.is_wintouched()

Return True if the specified window was modified since the last call to refresh(); otherwise return False.

window.keypad(flag)

If flag is True, escape sequences generated by some keys (keypad, function keys) will be interpreted by curses. If flag is False, escape sequences will be left as is in the input stream.

window.leaveok(flag)
If flag is True, cursor is left where it is on update, instead of being at “cursor position.” This reduces cursor movement where possible. If possible the cursor will be made invisible.

If flag is False, cursor will always be at “cursor position” after an update.

`window.move(new_y, new_x)`
Move cursor to \((\text{new}_y, \text{new}_x)\).

`window.mvderwin(y, x)`
Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

`window.mvwin(new_y, new_x)`
Move the window so its upper-left corner is at \((\text{new}_y, \text{new}_x)\).

`window.nodelay(flag)`
If flag is True, `getch()` will be non-blocking.

`window.notimeout(flag)`
If flag is True, escape sequences will not be timed out.

If flag is False, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

`window.noutrefresh()`
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call `doupdate()`.
window.**overlay**(destwin[, sminrow, smincol, dminrow, dmincol, 
dmaxrow, dmaxcol])

Overlay the window on top of `destwin`. The windows need not be 
the same size, only the overlapping region is copied. This copy is 
non-destructive, which means that the current background 
character does not overwrite the old contents of `destwin`.

To get fine-grained control over the copied region, the second form 
of `overlay()` can be used. `sminrow` and `smincol` are the upper-
left coordinates of the source window, and the other variables mark 
a rectangle in the destination window.

window.**overwrite**(destwin[, sminrow, smincol, dminrow, 
dmincol, dmaxrow, dmaxcol])

Overwrite the window on top of `destwin`. The windows need not be 
the same size, in which case only the overlapping region is copied. 
This copy is destructive, which means that the current background 
character overwrites the old contents of `destwin`.

To get fine-grained control over the copied region, the second form 
of `overwrite()` can be used. `sminrow` and `smincol` are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(file)

Write all data associated with the window into the provided file 
object. This information can be later retrieved using the `getwin()` 
function.

window.**redrawln**(beg, num)

Indicate that the `num` screen lines, starting at line `beg`, are 
corrupted and should be completely redrawn on the next
refresh() call.

window.redrawwin()

Touch the entire window, causing it to be completely redrawn on the next refresh() call.

window.refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with newpad(). The additional parameters are needed to indicate what part of the pad and screen are involved. pminrow and pmincol specify the upper left-hand corner of the rectangle to be displayed in the pad. sminrow, smincol, smaxrow, and smaxcol specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of pminrow, pmincol, sminrow, or smincol are treated as if they were zero.

window.resize(nlines, ncols)

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window’s data is filled with blanks that have the current background rendition (as set by bkgdset()) merged into them.

window.scroll([lines=1])

Scroll the screen or scrolling region upward by lines lines.
window. **scrollok**(*flag*)

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is **False**, the cursor is left on the bottom line. If *flag* is **True**, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok()**.

window. **setscrreg**(top, bottom)

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window. **standend**()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window. **standout**()

Turn on attribute `A_STANDOUT`.

window. **subpad**(begin_y, begin_x)
window. **subpad**(nlines, ncols, begin_y, begin_x)

Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

window. **subwin**(begin_y, begin_x)
window. **subwin**(nlines, ncols, begin_y, begin_x)

Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

By default, the sub-window will extend from the specified position to the lower right corner of the window.
window. **syncdown()**

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by **refresh()**, so it should almost never be necessary to call it manually.

window. **syncok(flag)**

If *flag* is **True**, then **syncup()** is called automatically whenever there is a change in the window.

window. **syncup()**

Touch all locations in ancestors of the window that have been changed in the window.

window. **timeout(delay)**

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and **getch()** will return **-1** if no input is waiting. If *delay* is positive, then **getch()** will block for *delay* milliseconds, and return **-1** if there is still no input at the end of that time.

window. **touchline(start, count[, changed])**

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (**changed**=**True**) or unchanged (**changed**=**False**).

window. **touchwin()**

Pretend the whole window has been changed, for purposes of drawing optimizations.

window. **untouchwin()**
Mark all lines in the window as unchanged since the last call to `refresh()`.

\[
\text{window.\texttt{vline}(ch, n)} \\
\text{window.\texttt{vline}(y, x, ch, n)}
\]

Display a vertical line starting at \((y, x)\) with length \(n\) consisting of the character \(ch\).
The `curses` module defines the following data members:

**curses.ERR**
Some curses routines that return an integer, such as `getch()`, return **ERR** upon failure.

**curses.OK**
Some curses routines that return an integer, such as `napms()`, return **OK** upon success.

**curses.version**
A bytes object representing the current version of the module. Also available as `__version__`.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ALTCCHARSET</td>
<td>Alternate character set mode</td>
</tr>
<tr>
<td>A_BLINK</td>
<td>Blink mode</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Bold mode</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Dim mode</td>
</tr>
<tr>
<td>A_INVIS</td>
<td>Invisible or blank mode</td>
</tr>
<tr>
<td>A_ITALIC</td>
<td>Italic mode</td>
</tr>
<tr>
<td>A_NORMAL</td>
<td>Normal attribute</td>
</tr>
<tr>
<td>A_PROTECT</td>
<td>Protected mode</td>
</tr>
<tr>
<td></td>
<td>Reverse background and foreground</td>
</tr>
<tr>
<td>A_REVERSE</td>
<td>colors</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>Standout mode</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underline mode</td>
</tr>
<tr>
<td>A_HORIZONTAL</td>
<td>Horizontal highlight</td>
</tr>
<tr>
<td>A_LEFT</td>
<td>Left highlight</td>
</tr>
<tr>
<td>A_LOW</td>
<td>Low highlight</td>
</tr>
<tr>
<td>A_RIGHT</td>
<td>Right highlight</td>
</tr>
<tr>
<td>A_TOP</td>
<td>Top highlight</td>
</tr>
<tr>
<td>A_VERTICAL</td>
<td>Vertical highlight</td>
</tr>
<tr>
<td>A_CHARTEXT</td>
<td>Bit-mask to extract a character</td>
</tr>
</tbody>
</table>

New in version 3.7: A_ITALIC was added.

Several constants are available to extract corresponding attributes returned by some methods.

<table>
<thead>
<tr>
<th>Bit-mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ATTRIBUTES</td>
<td>Bit-mask to extract attributes</td>
</tr>
<tr>
<td>A_CHARTEXT</td>
<td>Bit-mask to extract a character</td>
</tr>
<tr>
<td>A_COLOR</td>
<td>Bit-mask to extract color-pair field information</td>
</tr>
</tbody>
</table>

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

<table>
<thead>
<tr>
<th>Key constant</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_MIN</td>
<td>Minimum key value</td>
</tr>
<tr>
<td>KEY_BREAK</td>
<td>Break key (unreliable)</td>
</tr>
<tr>
<td>Key Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Down-arrow</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Up-arrow</td>
</tr>
<tr>
<td>KEY_LEFT</td>
<td>Left-arrow</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Right-arrow</td>
</tr>
<tr>
<td>KEY_HOME</td>
<td>Home key (upward+left arrow)</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Backspace (unreliable)</td>
</tr>
<tr>
<td>KEY_F0</td>
<td>Function keys. Up to 64 function keys are supported.</td>
</tr>
<tr>
<td>KEY_Fn</td>
<td>Value of function key n</td>
</tr>
<tr>
<td>KEY_DL</td>
<td>Delete line</td>
</tr>
<tr>
<td>KEY_IL</td>
<td>Insert line</td>
</tr>
<tr>
<td>KEY_DC</td>
<td>Delete character</td>
</tr>
<tr>
<td>KEY_IC</td>
<td>Insert char or enter insert mode</td>
</tr>
<tr>
<td>KEY_EIC</td>
<td>Exit insert char mode</td>
</tr>
<tr>
<td>KEY_CLEAR</td>
<td>Clear screen</td>
</tr>
<tr>
<td>KEY_EOS</td>
<td>Clear to end of screen</td>
</tr>
<tr>
<td>KEY_EOL</td>
<td>Clear to end of line</td>
</tr>
<tr>
<td>KEY_SF</td>
<td>Scroll 1 line forward</td>
</tr>
<tr>
<td>KEY_SR</td>
<td>Scroll 1 line backward (reverse)</td>
</tr>
<tr>
<td>KEY_NPAGE</td>
<td>Next page</td>
</tr>
<tr>
<td>KEY_PPAGE</td>
<td>Previous page</td>
</tr>
<tr>
<td>KEY_STAB</td>
<td>Set tab</td>
</tr>
<tr>
<td>KEY_CTAB</td>
<td>Clear tab</td>
</tr>
<tr>
<td>KEY_CATAB</td>
<td>Clear all tabs</td>
</tr>
<tr>
<td>Key</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>KEY_ENTER</td>
<td>Enter or send (unreliable)</td>
</tr>
<tr>
<td>KEY_SRESET</td>
<td>Soft (partial) reset (unreliable)</td>
</tr>
<tr>
<td>KEY_RESET</td>
<td>Reset or hard reset (unreliable)</td>
</tr>
<tr>
<td>KEY_PRINT</td>
<td>Print</td>
</tr>
<tr>
<td>KEY_LL</td>
<td>Home down or bottom (lower left)</td>
</tr>
<tr>
<td>KEY_A1</td>
<td>Upper left of keypad</td>
</tr>
<tr>
<td>KEY_A3</td>
<td>Upper right of keypad</td>
</tr>
<tr>
<td>KEY_B2</td>
<td>Center of keypad</td>
</tr>
<tr>
<td>KEY_C1</td>
<td>Lower left of keypad</td>
</tr>
<tr>
<td>KEY_C3</td>
<td>Lower right of keypad</td>
</tr>
<tr>
<td>KEY_BTAB</td>
<td>Back tab</td>
</tr>
<tr>
<td>KEY_BEG</td>
<td>Beg (beginning)</td>
</tr>
<tr>
<td>KEY_CANCEL</td>
<td>Cancel</td>
</tr>
<tr>
<td>KEY_CLOSE</td>
<td>Close</td>
</tr>
<tr>
<td>KEY_COMMAND</td>
<td>Cmd (command)</td>
</tr>
<tr>
<td>KEY_COPY</td>
<td>Copy</td>
</tr>
<tr>
<td>KEY_CREATE</td>
<td>Create</td>
</tr>
<tr>
<td>KEY_END</td>
<td>End</td>
</tr>
<tr>
<td>KEY_EXIT</td>
<td>Exit</td>
</tr>
<tr>
<td>KEY_FIND</td>
<td>Find</td>
</tr>
<tr>
<td>KEY_HELP</td>
<td>Help</td>
</tr>
<tr>
<td>KEY_MARK</td>
<td>Mark</td>
</tr>
<tr>
<td>KEY_MESSAGE</td>
<td>Message</td>
</tr>
<tr>
<td>KEY_MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>Key</td>
<td>Function</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>KEY_NEXT</td>
<td>Next</td>
</tr>
<tr>
<td>KEY_OPEN</td>
<td>Open</td>
</tr>
<tr>
<td>KEY_OPTIONS</td>
<td>Options</td>
</tr>
<tr>
<td>KEY_PREVIOUS</td>
<td>Prev (previous)</td>
</tr>
<tr>
<td>KEY_REDO</td>
<td>Redo</td>
</tr>
<tr>
<td>KEY_REFERENCE</td>
<td>Ref (reference)</td>
</tr>
<tr>
<td>KEY_REFRESH</td>
<td>Refresh</td>
</tr>
<tr>
<td>KEY_REPLACE</td>
<td>Replace</td>
</tr>
<tr>
<td>KEY_RESTART</td>
<td>Restart</td>
</tr>
<tr>
<td>KEY_RESUME</td>
<td>Resume</td>
</tr>
<tr>
<td>KEY_SAVE</td>
<td>Save</td>
</tr>
<tr>
<td>KEY_SBEG</td>
<td>Shifted Beg (beginning)</td>
</tr>
<tr>
<td>KEY_SCANCEL</td>
<td>Shifted Cancel</td>
</tr>
<tr>
<td>KEY_SCOMMAND</td>
<td>Shifted Command</td>
</tr>
<tr>
<td>KEY_SCOPY</td>
<td>Shifted Copy</td>
</tr>
<tr>
<td>KEY_SCREATE</td>
<td>Shifted Create</td>
</tr>
<tr>
<td>KEY_SDC</td>
<td>Shifted Delete char</td>
</tr>
<tr>
<td>KEYSDL</td>
<td>Shifted Delete line</td>
</tr>
<tr>
<td>KEY_SELECT</td>
<td>Select</td>
</tr>
<tr>
<td>KEY_SEND</td>
<td>Shifted End</td>
</tr>
<tr>
<td>KEY_SEOL</td>
<td>Shifted Clear line</td>
</tr>
<tr>
<td>KEY_SEXIT</td>
<td>Shifted Exit</td>
</tr>
<tr>
<td>KEY_SFIND</td>
<td>Shifted Find</td>
</tr>
<tr>
<td>KEY_SHELP</td>
<td>Shifted Help</td>
</tr>
<tr>
<td>Key Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>KEY_SHOME</td>
<td>Shifted Home</td>
</tr>
<tr>
<td>KEY_SIC</td>
<td>Shifted Input</td>
</tr>
<tr>
<td>KEY_SLEFT</td>
<td>Shifted Left arrow</td>
</tr>
<tr>
<td>KEY_SMMESSAGE</td>
<td>Shifted Message</td>
</tr>
<tr>
<td>KEY_SMOVE</td>
<td>Shifted Move</td>
</tr>
<tr>
<td>KEY_SNEXT</td>
<td>Shifted Next</td>
</tr>
<tr>
<td>KEY_SOPTIONS</td>
<td>Shifted Options</td>
</tr>
<tr>
<td>KEY_SPREVIOUS</td>
<td>Shifted Prev</td>
</tr>
<tr>
<td>KEY_SPRINT</td>
<td>Shifted Print</td>
</tr>
<tr>
<td>KEY_SREDO</td>
<td>Shifted Redo</td>
</tr>
<tr>
<td>KEY_SREPLACE</td>
<td>Shifted Replace</td>
</tr>
<tr>
<td>KEY_SRIGHT</td>
<td>Shifted Right arrow</td>
</tr>
<tr>
<td>KEY_SRSUME</td>
<td>Shifted Resume</td>
</tr>
<tr>
<td>KEY_SSAVE</td>
<td>Shifted Save</td>
</tr>
<tr>
<td>KEY_SSUSPEND</td>
<td>Shifted Suspend</td>
</tr>
<tr>
<td>KEY_SUNDO</td>
<td>Shifted Undo</td>
</tr>
<tr>
<td>KEY_SUSPEND</td>
<td>Suspend</td>
</tr>
<tr>
<td>KEY_UNDO</td>
<td>Undo</td>
</tr>
<tr>
<td>KEY_MOUSE</td>
<td>Mouse event has occurred</td>
</tr>
<tr>
<td>KEY_RESIZE</td>
<td>Terminal resize event</td>
</tr>
<tr>
<td>KEY_MAX</td>
<td>Maximum key value</td>
</tr>
</tbody>
</table>

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to
KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

<table>
<thead>
<tr>
<th>Keycap</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>KEY_IC</td>
</tr>
<tr>
<td>Delete</td>
<td>KEY_DC</td>
</tr>
<tr>
<td>Home</td>
<td>KEY_HOME</td>
</tr>
<tr>
<td>End</td>
<td>KEY_END</td>
</tr>
<tr>
<td>Page Up</td>
<td>KEY_PPAGE</td>
</tr>
<tr>
<td>Page Down</td>
<td>KEY_NPAGE</td>
</tr>
</tbody>
</table>

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

**Note:** These are available only after `initscr()` has been called.

<table>
<thead>
<tr>
<th>ACS code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS_BBSS</td>
<td>alternate name for upper right corner</td>
</tr>
<tr>
<td>ACS_BLOCK</td>
<td>solid square block</td>
</tr>
<tr>
<td>ACS_BOARD</td>
<td>board of squares</td>
</tr>
<tr>
<td>ACS_BSBS</td>
<td>alternate name for horizontal line</td>
</tr>
<tr>
<td>ACS_BSSB</td>
<td>alternate name for upper left corner</td>
</tr>
<tr>
<td>ACS_BSSS</td>
<td>alternate name for top tee</td>
</tr>
<tr>
<td>ACS_BTEE</td>
<td>bottom tee</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>ACS_BULLET</td>
<td>bullet</td>
</tr>
<tr>
<td>ACS_CKBOARD</td>
<td>checker board (stipple)</td>
</tr>
<tr>
<td>ACS_DARROW</td>
<td>arrow pointing down</td>
</tr>
<tr>
<td>ACS_DEGREE</td>
<td>degree symbol</td>
</tr>
<tr>
<td>ACS_DIAMOND</td>
<td>diamond</td>
</tr>
<tr>
<td>ACS_GEQUAL</td>
<td>greater-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_HLINE</td>
<td>horizontal line</td>
</tr>
<tr>
<td>ACS_LANTERN</td>
<td>lantern symbol</td>
</tr>
<tr>
<td>ACS_LARROW</td>
<td>left arrow</td>
</tr>
<tr>
<td>ACS_LEQUAL</td>
<td>less-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_LLCORNER</td>
<td>lower left-hand corner</td>
</tr>
<tr>
<td>ACS_LRCORNER</td>
<td>lower right-hand corner</td>
</tr>
<tr>
<td>ACS_LTEE</td>
<td>left tee</td>
</tr>
<tr>
<td>ACS_NEQUAL</td>
<td>not-equal sign</td>
</tr>
<tr>
<td>ACS_PI</td>
<td>letter pi</td>
</tr>
<tr>
<td>ACS_PLMINUS</td>
<td>plus-or-minus sign</td>
</tr>
<tr>
<td>ACS_PLUS</td>
<td>big plus sign</td>
</tr>
<tr>
<td>ACS_RARROW</td>
<td>right arrow</td>
</tr>
<tr>
<td>ACS_RTEE</td>
<td>right tee</td>
</tr>
<tr>
<td>ACS_S1</td>
<td>scan line 1</td>
</tr>
<tr>
<td>ACS_S3</td>
<td>scan line 3</td>
</tr>
<tr>
<td>ACS_S7</td>
<td>scan line 7</td>
</tr>
<tr>
<td>ACS_S9</td>
<td>scan line 9</td>
</tr>
<tr>
<td>ACS_SBBS</td>
<td>alternate name for lower right corner</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>ACS_SBSB</td>
<td>alternate name for vertical line</td>
</tr>
<tr>
<td>ACS_SBSS</td>
<td>alternate name for right tee</td>
</tr>
<tr>
<td>ACS_SSBB</td>
<td>alternate name for lower left corner</td>
</tr>
<tr>
<td>ACS_SSBS</td>
<td>alternate name for bottom tee</td>
</tr>
<tr>
<td>ACS_SSSB</td>
<td>alternate name for left tee</td>
</tr>
<tr>
<td>ACS_SSSS</td>
<td>alternate name for crossover or big plus</td>
</tr>
<tr>
<td>ACS_STERLING</td>
<td>pound sterling</td>
</tr>
<tr>
<td>ACS_TTEE</td>
<td>top tee</td>
</tr>
<tr>
<td>ACS_UARROW</td>
<td>up arrow</td>
</tr>
<tr>
<td>ACS_ULCORNER</td>
<td>upper left corner</td>
</tr>
<tr>
<td>ACS_URCORNER</td>
<td>upper right corner</td>
</tr>
<tr>
<td>ACS_VLINE</td>
<td>vertical line</td>
</tr>
</tbody>
</table>

The following table lists the predefined colors:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_BLACK</td>
<td>Black</td>
</tr>
<tr>
<td>COLOR_BLUE</td>
<td>Blue</td>
</tr>
<tr>
<td>COLOR_CYAN</td>
<td>Cyan (light greenish blue)</td>
</tr>
<tr>
<td>COLOR_GREEN</td>
<td>Green</td>
</tr>
<tr>
<td>COLOR_MAGENTA</td>
<td>Magenta (purplish red)</td>
</tr>
<tr>
<td>COLOR_RED</td>
<td>Red</td>
</tr>
<tr>
<td>COLOR_WHITE</td>
<td>White</td>
</tr>
<tr>
<td>COLOR_YELLOW</td>
<td>Yellow</td>
</tr>
</tbody>
</table>
curses.textpad — Text input widget for curses programs

The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

```
curses.textpad.rectangle(win, uly, ulx, lry, lrx)
```

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.
Textbox objects

You can instantiate a `Textbox` object as follows:

```python
class curses.textpad.Textbox(win)
```

Return a textbox widget object. The `win` argument should be a curses `window` object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates \((0, 0)\). The instance's `stripspaces` flag is initially on.

`Textbox` objects have the following methods:

**edit([validator])**

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If `validator` is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` attribute.

**do_command(ch)**

Process a single command keystroke. Here are the supported special keystrokes:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-A</td>
<td>Go to left edge of window.</td>
</tr>
<tr>
<td>Control-B</td>
<td>Cursor left, wrapping to previous line if appropriate.</td>
</tr>
<tr>
<td>Control-D</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Control-E</td>
<td>Go to right edge (stripspaces off) or end of line (stripspaces on).</td>
</tr>
<tr>
<td>Control-F</td>
<td>Cursor right, wrapping to next line when appropriate.</td>
</tr>
<tr>
<td>Control-G</td>
<td>Terminate, returning the window contents.</td>
</tr>
<tr>
<td>Control-H</td>
<td>Delete character backward.</td>
</tr>
<tr>
<td>Control-J</td>
<td>Terminate if the window is 1 line, otherwise insert newline.</td>
</tr>
<tr>
<td>Control-K</td>
<td>If line is blank, delete it, otherwise clear to end of line.</td>
</tr>
<tr>
<td>Control-L</td>
<td>Refresh screen.</td>
</tr>
<tr>
<td>Control-N</td>
<td>Cursor down; move down one line.</td>
</tr>
<tr>
<td>Control-O</td>
<td>Insert a blank line at cursor location.</td>
</tr>
<tr>
<td>Control-P</td>
<td>Cursor up; move up one line.</td>
</tr>
</tbody>
</table>

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_LEFT</td>
<td>Control-B</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Control-F</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Control-P</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Control-N</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Control-h</td>
</tr>
</tbody>
</table>

All other keystrokes are treated as a command to insert the
given character and move right (with line wrapping).

**gather()**

Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

**stripspaces**

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.
The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td></td>
</tr>
<tr>
<td>SOH</td>
<td>Start of heading, console interrupt</td>
</tr>
<tr>
<td>STX</td>
<td>Start of text</td>
</tr>
<tr>
<td>ETX</td>
<td>End of text</td>
</tr>
<tr>
<td>EOT</td>
<td>End of transmission</td>
</tr>
<tr>
<td>ENQ</td>
<td>Enquiry, goes with ACK flow control</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>BEL</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>TAB</td>
<td>Tab</td>
</tr>
<tr>
<td>HT</td>
<td>Alias for TAB: “Horizontal tab”</td>
</tr>
<tr>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>NL</td>
<td>Alias for LF: “New line”</td>
</tr>
<tr>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>Shift-out, begin alternate character set</td>
</tr>
<tr>
<td>SI</td>
<td>Shift-in, resume default character set</td>
</tr>
<tr>
<td>DLE</td>
<td>Data-link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>XON, for flow control</td>
</tr>
<tr>
<td>DC2</td>
<td>Device control 2, block-mode flow control</td>
</tr>
<tr>
<td>DC3</td>
<td>XOFF, for flow control</td>
</tr>
<tr>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>NAK</td>
<td>Negative acknowledgement</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>End transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>Record separator, block-mode terminator</td>
</tr>
<tr>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.
The module supplies the following functions, patterned on those in the standard C library:

- `curses.ascii.isalnum(c)`
  Checks for an ASCII alphanumerical character; it is equivalent to `isalpha(c)` or `isdigit(c)`.

- `curses.ascii.isalpha(c)`
  Checks for an ASCII alphabetic character; it is equivalent to `isupper(c)` or `islower(c)`.

- `curses.ascii.isascii(c)`
  Checks for a character value that fits in the 7-bit ASCII set.

- `curses.ascii.isblank(c)`
  Checks for an ASCII whitespace character; space or horizontal tab.

- `curses.ascii.iscntrl(c)`
  Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

- `curses.ascii.isdigit(c)`
  Checks for an ASCII decimal digit, '0' through '9'. This is equivalent to `c in string.digits`.

- `curses.ascii.isgraph(c)`
  Checks for ASCII any printable character except space.

- `curses.ascii.islower(c)`
  Checks for an ASCII lower-case character.

- `curses.ascii.isprint(c)`
  Checks for any ASCII printable character including space.
curses.ascii.\texttt{ispunct}(c)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.\texttt{isspace}(c)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.\texttt{isupper}(c)
Checks for an ASCII uppercase letter.

curses.ascii.\texttt{isxdigit}(c)
Checks for an ASCII hexadecimal digit. This is equivalent to \texttt{c in string.hexdigits}.

curses.ascii.\texttt{isctrl}(c)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.\texttt{ismeta}(c)
Checks for a non-ASCII character (ordinal values \texttt{0x80} and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function \texttt{ord()}.  

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine’s character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.\texttt{ascii}(c)
Return the ASCII value corresponding to the low 7 bits of c.

curses.ascii.ctrl(c)
    Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.alt(c)
    Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.uncctrl(c)
    Return a string representation of the ASCII character c. If c is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret ('^') followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is '^?'. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and '!' prepended to the result.

curses.ascii.controlnames
    A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic SP for the space character.
curses.panel — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.
Functions

The module `curses.panel` defines the following functions:

- `curses.panel.bottom_panel()`
  Returns the bottom panel in the panel stack.

- `curses.panel.new_panel(win)`
  Returns a panel object, associating it with the given window `win`. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

- `curses.panel.top_panel()`
  Returns the top panel in the panel stack.

- `curses.panel.update_panels()`
  Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you'll have to do this yourself.
Panel Objects

Panel objects, as returned by \texttt{new\_panel()} above, are windows with a stacking order. There’s always a window associated with a panel which determines the content, while the panel methods are responsible for the window’s depth in the panel stack.

Panel objects have the following methods:

\begin{itemize}
  \item \texttt{Panel.\texttt{above}()} \\
  \hspace{1em} Returns the panel above the current panel.

  \item \texttt{Panel.\texttt{below}()} \\
  \hspace{1em} Returns the panel below the current panel.

  \item \texttt{Panel.\texttt{bottom}()} \\
  \hspace{1em} Push the panel to the bottom of the stack.

  \item \texttt{Panel.\texttt{hidden}()} \\
  \hspace{1em} Returns \texttt{True} if the panel is hidden (not visible), \texttt{False} otherwise.

  \item \texttt{Panel.\texttt{hide}()} \\
  \hspace{1em} Hide the panel. This does not delete the object, it just makes the window on screen invisible.

  \item \texttt{Panel.\texttt{move}(y, x)} \\
  \hspace{1em} Move the panel to the screen coordinates \((y, x)\).

  \item \texttt{Panel.\texttt{replace}(win)} \\
  \hspace{1em} Change the window associated with the panel to the window \texttt{win}.

  \item \texttt{Panel.\texttt{set\_userptr}(obj)}
\end{itemize}
Set the panel's user pointer to `obj`. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

`Panel.show()`
Display the panel (which might have been hidden).

`Panel.top()`
Push panel to the top of the stack.

`Panel.userptr()`
Returns the user pointer for the panel. This might be any Python object.

`Panel.window()`
Returns the window object associated with the panel.
platform — Access to underlying platform’s identifying data

Source code: Lib/platform.py

Note: Specific platforms listed alphabetically, with Linux included in the Unix section.
Cross Platform

platform.architecture(executable=sys.executable, bits='', linkage='')

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple (bits, linkage) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as '', the sizeof(pointer) (or sizeof(long) on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system’s file command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note: On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures. To get at the “64-bitness” of the current interpreter, it is more reliable to query the sys.maxsize attribute:

```
is_64bits = sys.maxsize > 2**32
```

platform.machine()

Returns the machine type, e.g. 'i386'. An empty string is
returned if the value cannot be determined.

**platform.node()**

Returns the computer’s network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

**platform.platform(aliased=0, terse=0)**

Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended.

If `aliased` is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The `system_alias()` function is used to implement this.

Setting `terse` to true causes the function to return only the absolute minimum information needed to identify the platform.

**platform.processor()**

Returns the (real) processor name, e.g. `'amdk6'`.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for `machine()`. NetBSD does this.

**platform.python_build()**

Returns a tuple `(buildno, builddate)` stating the Python build number and date as strings.

**platform.python_compiler()**
Returns a string identifying the compiler used for compiling Python.

```
platform.python_branch()
```

Returns a string identifying the Python implementation SCM branch.

```
platform.python_implementation()
```


```
platform.python_revision()
```

Returns a string identifying the Python implementation SCM revision.

```
platform.python_version()
```

Returns the Python version as string `major.minor.patchlevel`.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to 0).

```
platform.python_version_tuple()
```

Returns the Python version as tuple `(major, minor, patchlevel)` of strings.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to `'0'`).

```
platform.release()
```

Returns the system's release, e.g. `'2.2.0'` or `'NT'` An empty string is returned if the value cannot be determined.

```
platform.system()
```
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

**platform.system_alias**(system, release, version)

Returns (system, release, version) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

**platform.version()**

Returns the system's release version, e.g. '#3 on degas'. An empty string is returned if the value cannot be determined.

**platform.uname()**

Fairly portable uname interface. Returns a namedtuple() containing six attributes: system, node, release, version, machine, and processor.

Note that this adds a sixth attribute (processor) not present in the os.uname() result. Also, the attribute names are different for the first two attributes; os.uname() names them sysname and nodename.

Entries which cannot be determined are set to ''.

*Changed in version 3.3:* Result changed from a tuple to a namedtuple.
Java Platform

platform.java_ver(release='', vendor='', vminfo=('', '', ''), osinfo=('', '', ''))

Version interface for Jython.

Returns a tuple (release, vendor, vminfo, osinfo) with
vminfo being a tuple (vm_name, vm_release, vm_vendor)
and osinfo being a tuple (os_name, os_version, os_arch).
Values which cannot be determined are set to the defaults given as
parameters (which all default to '').
Windows Platform

platform.\texttt{win32\_ver}(\texttt{release=", version=", csd=", ptype=\"")

Get additional version information from the Windows Registry and return a tuple \texttt{(release, version, csd, ptype)} referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint: \texttt{ptype} is 'Uniprocessor Free' on single processor NT machines and 'Multiprocessor Free' on multi processor machines. The 'Free' refers to the OS version being free of debugging code. It could also state 'Checked' which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

\begin{quote}
\textbf{Note:} This function works best with Mark Hammond's \texttt{win32all} package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.
\end{quote}

Win95/98 specific

platform.\texttt{popen}(\texttt{cmd, mode='r', bufsize=-1})

Portable \texttt{popen()} interface. Find a working popen implementation preferring \texttt{win32pipe.popen()}. On Windows NT, \texttt{win32pipe.popen()} should work; on Windows 9x it hangs due to bugs in the MS C library.

\textit{Deprecated since version 3.3:} This function is obsolete. Use the \texttt{subprocess} module. Check especially the \texttt{Replacing Older}
Functions with the subprocess Module section.
Mac OS Platform

platform.mach_ver(release='', versioninfo=('', '', ''), machine= '')

Get Mac OS version information and return it as tuple (release, versioninfo, machine) with versioninfo being a tuple (version, dev_stage, non_release_version).

Entries which cannot be determined are set to ''. All tuple entries are strings.
Unix Platforms

platform.dist(distname='', version='', id='', supported_dists= ('SuSE', 'debian', 'redhat', 'mandrake', ...))

This is another name for linux_distribution().

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.linux_distribution(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...), full_distribution_name=1)

Tries to determine the name of the Linux OS distribution name.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If full_distribution_name is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from supported_dists is used.

Returns a tuple (distname, version, id) which defaults to the args given as parameters. id is the item in parentheses after the version number. It is usually the version codename.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.libc_ver(executable=sys.executable, lib='', version='', chunksize=16384)

Tries to determine the libc version against which the file executable
(defaults to the Python interpreter) is linked. Returns a tuple of strings \( \text{lib, version} \) which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using \texttt{gcc}.

The file is read and scanned in chunks of \textit{chunksize} bytes.
**errno — Standard errno system symbols**

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be pretty all-inclusive.

`errno.errorcode`

Dictionary providing a mapping from the `errno` value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to 'EPERM'.

To translate a numeric error code to an error message, use `os.strerror()`.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

- **errno.**`EPERM`
  - Operation not permitted

- **errno.**`ENOENT`
  - No such file or directory

- **errno.**`ESRCH`
  - No such process

- **errno.**`EINTR`
Interrupted system call.

See also: This error is mapped to the exception InterruptedError.

errno. **EIO**
I/O error

errno. **ENXIO**
No such device or address

errno. **E2BIG**
Arg list too long

errno. **ENOEXEC**
Exec format error

errno. **EBADF**
Bad file number

errno. **ECHILD**
No child processes

errno. **EAGAIN**
Try again

errno. **ENOMEM**
Out of memory

errno. **EACCES**
Permission denied

errno. **EFAULT**
Bad address

`errno. ENOTBLK`
Block device required

`errno. EBUSY`
Device or resource busy

`errno. EEXIST`
File exists

`errno. EXDEV`
Cross-device link

`errno. ENODEV`
No such device

`errno. ENOTDIR`
Not a directory

`errno. EISDIR`
Is a directory

`errno. EINVAL`
Invalid argument

`errno. ENFILE`
File table overflow

`errno. EMFILE`
Too many open files

`errno. ENOTTY`
Not a typewriter
errno. **ETXTBSY**
   Text file busy

errno. **EFBIG**
   File too large

errno. **ENOSPC**
   No space left on device

errno. **ESPIPE**
   Illegal seek

errno. **EROFS**
   Read-only file system

errno. **EMLINK**
   Too many links

errno. **EPIPE**
   Broken pipe

errno. **EDOM**
   Math argument out of domain of func

errno. **ERANGE**
   Math result not representable

errno. **EDEADLK**
   Resource deadlock would occur

errno. **ENAMETOOLONG**
   File name too long

errno. **ENOLCK**
No record locks available

errno. **ENOSYS**
Function not implemented

errno. **ENOTEMPTY**
Directory not empty

errno. **ELOOP**
Too many symbolic links encountered

errno. **EWOULDBLOCK**
Operation would block

errno. **ENOMSG**
No message of desired type

errno. **EIDRM**
Identifier removed

errno. **ECHRNG**
Channel number out of range

errno. **EL2NSYNC**
Level 2 not synchronized

errno. **EL3HLT**
Level 3 halted

errno. **EL3RST**
Level 3 reset

errno. **ELNRNG**
Link number out of range
errno. **EUNATCH**
Protocol driver not attached

errno. **ENOCSI**
No CSI structure available

errno. **EL2HLT**
Level 2 halted

errno. **EBADE**
Invalid exchange

errno. **EBADR**
Invalid request descriptor

errno. **EXFULL**
Exchange full

errno. **ENOANO**
No anode

errno. **EBADRQC**
Invalid request code

errno. **EBADSLT**
Invalid slot

errno. **EDEADLOCK**
File locking deadlock error

errno. **EBFONT**
Bad font file format

errno. **ENOSTR**
Device not a stream

errno. **ENODATA**
No data available

errno. **ETIME**
Timer expired

errno. **ENOSR**
Out of streams resources

errno. **ENONET**
Machine is not on the network

errno. **ENOPKG**
Package not installed

errno. **EREMOTE**
Object is remote

errno. **ENOLINK**
Link has been severed

errno. **EADV**
Advertise error

errno. **ESRMNT**
Srmount error

errno. **ECOMM**
Communication error on send

errno. **EPROTO**
Protocol error
errno. **EMULTIHOP**
Multihop attempted

errno. **EDOTDOT**
RFS specific error

errno. **EBADMSG**
Not a data message

errno. **EOVERFLOW**
Value too large for defined data type

errno. **ENOTUNIQ**
Name not unique on network

errno. **EBADFD**
File descriptor in bad state

errno. **EREMCHG**
Remote address changed

errno. **ELIBACC**
Can not access a needed shared library

errno. **ELIBBAD**
Accessing a corrupted shared library

errno. **ELIBSCN**
.lib section in a.out corrupted

errno. **ELIBMAX**
Attempting to link in too many shared libraries

errno. **ELIBEXEC**
Cannot exec a shared library directly

errio. EILSEQ
Illegal byte sequence

errio. ERESTART
Interrupted system call should be restarted

errio. ESTRPIPE
Streams pipe error

errio. EUSERS
Too many users

errio. ENOTSOCK
Socket operation on non-socket

errio. EDESTADDRREQ
Destination address required

errio. EMSGSIZE
Message too long

errio. EPROTOTYPE
Protocol wrong type for socket

errio. ENOPROTOOPT
Protocol not available

errio. EPROTONOSUPPORT
Protocol not supported

errio. ESOCKTNOSUPPORT
Socket type not supported
errno. **EOPNOTSUPP**
   Operation not supported on transport endpoint

errno. **EPFNOSUPPORT**
   Protocol family not supported

errno. **EAFNOSUPPORT**
   Address family not supported by protocol

errno. **EADDRINUSE**
   Address already in use

errno. **EADDRNOTAVAIL**
   Cannot assign requested address

errno. **ENETDOWN**
   Network is down

errno. **ENETUNREACH**
   Network is unreachable

errno. **ENETRESET**
   Network dropped connection because of reset

errno. **ECONNABORTED**
   Software caused connection abort

errno. **ECONNRESET**
   Connection reset by peer

errno. **ENOBUFFS**
   No buffer space available

errno. **EISCONN**
Transport endpoint is already connected

errno. **ENOTCONN**
Transport endpoint is not connected

errno. **ESHUTDOWN**
Cannot send after transport endpoint shutdown

errno. **ETOOMANYREFS**
Too many references: cannot splice

errno. **ETIMEDOUT**
Connection timed out

errno. **ECONNREFUSED**
Connection refused

errno. **EHOSTDOWN**
Host is down

errno. **EHOSTUNREACH**
No route to host

errno. **EALREADY**
Operation already in progress

errno. **EINPROGRESS**
Operation now in progress

errno. **ESTALE**
Stale NFS file handle

errno. **EUCLean**
Structure needs cleaning
errno. **ENOTNAM**  
Not a XENIX named type file

errno. **ENAVAIL**  
No XENIX semaphores available

errno. **EISNAM**  
Is a named type file

errno. **EREMOTEIO**  
Remote I/O error

errno. **EDQUOT**  
Quota exceeded
ctypes — A foreign function library for Python

cTypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.
ctypes tutorial

Note: The code samples in this tutorial use doctest to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Some code samples reference the ctypes c_int type. On platforms where sizeof(long) == sizeof(int) it is an alias to c_long. So, you should not be confused if c_long is printed if you would expect c_int — they are actually the same type.

Loading dynamic link libraries

ctypes exports the cdll, and on Windows windll and oledll objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. cdll loads libraries which export functions using the standard cdecl calling convention, while windll libraries call functions using the stdcall calling convention. oledll also uses the stdcall calling convention, and assumes the functions return a Windows HRESULT error code. The error code is used to automatically raise an OSError exception when the function call fails.

Changed in version 3.3: Windows errors used to raise WindowsError, which is now an alias of OSError.

Here are some examples for Windows. Note that msvcrtn is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:
Windows appends the usual `.dll` file suffix automatically.

**Note:** Accessing the standard C library through `cdll.msvcr` will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the `msvcrt` module.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access cannot be used to load libraries. Either the `LoadLibrary()` method of the dll loaders should be used, or you should load the library by creating an instance of `CDLL` by calling the constructor:

```python
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
```
Note that win32 system dlls like kernel32 and user32 often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an \texttt{W} appended to the name, while the ANSI version is exported with an \texttt{A} appended to the name. The win32 GetModuleHandle function, which returns a \textit{module handle} for a given module name, has the following C prototype, and a macro is used to expose one of them as \texttt{GetModuleHandle} depending on whether UNICODE is defined or not:

\begin{verbatim}
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
\end{verbatim}

\texttt{windll} does not try to select one of them by magic, you must access the version you need by specifying \texttt{GetModuleHandleA} or \texttt{GetModuleHandleW} explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren’t valid Python identifiers, like "??2@YAPAXI@Z". In this case you have to use \texttt{getattr()} to retrieve the function:
On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```python
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
```  

```python
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
```  

### Calling functions

You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the Unix epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (`None` should be used as the NULL pointer):

```python
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
```
Note: `ctypes` may raise a `ValueError` after calling the function, if it detects that an invalid number of arguments were passed. This behavior should not be relied upon. It is deprecated in 3.6.2, and will be removed in 3.7.

`ValueError` is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```python
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments
```

```python
>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, `ctypes` uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```python
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000
```
There are, however, enough ways to crash Python with ctypes, so you should be careful anyway. The faulthandler module can be helpful in debugging crashes (e.g. from segmentation faults produced by erroneous C library calls).

None, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (char * or wchar_t *). Python integers are passed as the platforms default C int type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about ctypes data types.

**Fundamental data types**

ctypes defines a number of primitive C compatible data types:

<table>
<thead>
<tr>
<th>ctypes type</th>
<th>C type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_bool</td>
<td>_Bool</td>
<td>bool (1)</td>
</tr>
<tr>
<td>c_char</td>
<td>char</td>
<td>1-character bytes object</td>
</tr>
<tr>
<td>c_wchar</td>
<td>wchar_t</td>
<td>1-character string</td>
</tr>
<tr>
<td>c_byte</td>
<td>char</td>
<td>int</td>
</tr>
<tr>
<td>c_ubyte</td>
<td>unsigned char</td>
<td>int</td>
</tr>
<tr>
<td>c_short</td>
<td>short</td>
<td>int</td>
</tr>
<tr>
<td>c_ushort</td>
<td>unsigned short</td>
<td>int</td>
</tr>
<tr>
<td>Python Type</td>
<td>C Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>c_int</code></td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td><code>c_uint</code></td>
<td>unsigned int</td>
<td>int</td>
</tr>
<tr>
<td><code>c_long</code></td>
<td>long</td>
<td>int</td>
</tr>
<tr>
<td><code>c_ulong</code></td>
<td>unsigned long</td>
<td>int</td>
</tr>
<tr>
<td><code>c_ulonglong</code></td>
<td>__int64 or long long</td>
<td>int</td>
</tr>
<tr>
<td><code>c_ulonglong</code></td>
<td>unsigned __int64 or unsigned long long</td>
<td>int</td>
</tr>
<tr>
<td><code>c_size_t</code></td>
<td>size_t</td>
<td>int</td>
</tr>
<tr>
<td><code>c_ssize_t</code></td>
<td>ssize_t or Py_ssize_t</td>
<td>int</td>
</tr>
<tr>
<td><code>c_float</code></td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td><code>c_double</code></td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td><code>c_longdouble</code></td>
<td>long double</td>
<td>float</td>
</tr>
<tr>
<td><code>c_char_p</code></td>
<td>char * (NUL terminated)</td>
<td>bytes object or None</td>
</tr>
<tr>
<td><code>c_wchar_p</code></td>
<td>wchar_t * (NUL terminated)</td>
<td>string or None</td>
</tr>
<tr>
<td><code>c_void_p</code></td>
<td>void *</td>
<td>int or None</td>
</tr>
</tbody>
</table>

1. The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```python
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
```
Since these types are mutable, their value can also be changed afterwards:

```python
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>> 
```

Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the memory location they point to, not the contents of the memory block (of course not, because Python bytes objects are immutable):

```python
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s.value)  # the memory location has changed
"Hi, there"
>>> print(c_s)
 # first object is unchanged
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)  # first object is unchanged
Hello, World
>>> 
```
You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer()` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```python
>>> from ctypes import *
>>> p = create_string_buffer(3)  # create
>>> print(sizeof(p), repr(p.raw)) 3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")  # create
>>> print(sizeof(p), repr(p.raw)) 6 b'Hello\x00'
>>> print(repr(p.value)) b'Hello'
>>> p = create_string_buffer(b"Hello", 10)  # create
>>> print(sizeof(p), repr(p.raw)) 10 b'Hello\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw)) 10 b'Hi\x00lo\x00\x00\x00\x00\x00\x00\x00\x00'
```  

The `create_string_buffer()` function replaces the `c_buffer()` function (which is still available as an alias), as well as the `c_string()` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the `create_unicode_buffer()` function.

**Calling functions, continued**

Note that printf prints to the real standard output channel, *not* to
`sys.stdout`, so these examples will only work at the console prompt, not from within `IDLE` or `PythonWin`:

```python
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don
```

As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```python
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
```

**Calling functions with your own custom data types**

You can also customize `ctypes` argument conversion to allow instances of your own classes be used as function arguments.
**ctypes** looks for an _as_parameter_ attribute and uses this as the function argument. Of course, it must be one of integer, string, or bytes:

```python
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
```

If you don't want to store the instance's data in the _as_parameter_ instance variable, you could define a *property* which makes the attribute available on request.

### Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```python
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_
>>> >>> printf(b"String '%s', Int %d, Double %f\n", b"Hi"
String 'Hi', Int 10, Double 2.200000
```
Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```python
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>> 
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a `ctypes` instance, or an object with an `_as_parameter_` attribute.

**Return types**

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the `restype` attribute of the function object.
Here is a more advanced example, it uses the **strchr** function, which expects a string pointer and a char, and returns a pointer to a string:

```python
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p  # c_char_p is a pointer
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```python
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the `integer` the C function
returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```python
>>> GetModuleHandle = windll.kernel32.GetModuleHandle
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
... >>> GetModuleHandle.retype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not
```

WinError is a function which will call Windows FormatMessage() api to get the string representation of an error code, and returns an exception. WinError takes an optional error code parameter, if no one is used, it calls GetLastError() to retrieve it.

Please note that a much more powerful error checking mechanism is available through the errcheck attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)
Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the `byref()` function which is used to pass parameters by reference. The same effect can be achieved with the `pointer()` function, although `pointer()` does a lot more work since it constructs a real pointer object, so it is faster to use `byref()` if you don’t need the pointer object in Python itself:

```python
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc sscanf(b"1 3.14 Hello", b"%d %f %s", ...
    byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

**Structures and unions**

Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of 2-tuples, containing a *field name* and a *field type*.

The field type must be a `ctypes` type like `c_int`, or any other derived `ctypes` type: structure, union, array, pointer.
Here is a simple example of a POINT structure, which contains two integers named \( x \) and \( y \), and also shows how to initialize a structure in the constructor:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [('x', c_int),
...                 ('y', c_int)]
...  
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many initializers
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named upperleft and lowerright:

```python
>>> class RECT(Structure):
...     _fields_ = [('upperleft', POINT),
...                 ('lowerright', POINT)]
...  
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
```
Nested structures can also be initialized in the constructor in several ways:

```python
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field descriptors can be retrieved from the class, they are useful for debugging because they can provide useful information:

```python
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
```

**Warning:** `ctypes` does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

**Structure/union alignment and byte order**

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a `__pack__` class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

`ctypes` uses the native byte order for Structures and Unions. To build
structures with non-native byte order, you can use one of the 
BigEndianStructure, LittleEndianStructure, 
BigEndianUnion, and LittleEndianUnion base classes. These 
classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit 
fields are only possible for integer fields, the bit width is specified as 
the third item in the _fields_ tuples:

```python
>>> class Int(Structure):
...     _fields_ = [('first_16', c_int, 16),
...                 ('second_16', c_int, 16)]
...

>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>

.Arrays

Arrays are sequences, containing a fixed number of instances of the 
same type.

The recommended way to create array types is by multiplying a data 
type with a positive integer:

```
TenPointsArrayType = POINT * 10
```
>>> from ctypes import *
>>> class POINT(Structure):
...   _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...   _fields_ = [('a', c_int),
...     ('b', c_float),
...     ('point_array', POINT * 4)]
>>> print(len(MyStruct().point_array))
4

Instances are created in the usual way, by calling the class:

```python
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```python
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
... 1 2 3 4 5 6 7 8 9 10
```
Pointer instances are created by calling the `pointer()` function on a `ctypes` type:

```python
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)

>>> pi.contents
```

```python
c_long(42)
```

Pointer instances have a `contents` attribute which returns the object to which the pointer points, the `i` object above:

```python
>>> pi.contents
```

```python
c_long(42)
```

Note that `ctypes` does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```python
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
```

Assigning another `c_int` instance to the pointer’s contents attribute would cause the pointer to point to the memory location where this is stored:

```python
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
```

```python
c_long(99)
```
Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>> 
```

Assigning to an integer index changes the pointed to value:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>> 
```

It is also possible to use indexes different from 0, but you must know what you’re doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you know that the pointer actually points to an array instead of a single item.

Behind the scenes, the `pointer()` function does more than simply create pointer instances, it has to create pointer types first. This is done with the `POINTER()` function, which accepts any `ctypes` type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
```
Calling the pointer type without an argument creates a NULL pointer. NULL pointers have a False boolean value:

```python
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
```  

`ctypes` checks for NULL when dereferencing pointers (but dereferencing invalid non-NULL pointers would crash Python):

```python
>>> null_ptr[0]
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
```  

```python
>>> null_ptr[0] = 1234
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
```  

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the argtypes list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for
POINTER(c_int), ctypes accepts an array of c_int:

```python
>>> class Bar(Structure):
    ...     _fields_ = ["count", c_int], ("values", POINTER(c_int))
    ...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
    ...     print(bar.values[i])
    ...
1
2
3
```

In addition, if a function argument is explicitly declared to be a pointer type (such as POINTER(c_int)) in argtypes, an object of the pointed type (c_int in this case) can be passed to the function. ctypes will apply the required byref() conversion in this case automatically.

To set a POINTER type field to NULL, you can assign None:

```python
>>> bar.values = None
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. ctypes provides a cast() function which can be used in the same way. The Bar structure defined above accepts POINTER(c_int) pointers or c_int arrays for its values field, but not instances of other types:
For these cases, the `cast()` function is handy.

The `cast()` function can be used to cast a ctypes instance into a pointer to a different ctypes data type. `cast()` takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>

So, `cast()` can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
```

Incomplete Types

`Incomplete Types` are structures, unions or arrays whose members are
not yet specified. In C, they are specified by forward declarations, which are defined later:

```c
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```python
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined

>>> 
```

because the new `class cell` is not available in the class statement itself. In ctypes, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```python
>>> from ctypes import *

>>> class cell(Structure):
...     pass
...

>>> cell._fields_ = [("name", c_char_p),
...                   ("next", POINTER(cell))]

>>> 
```
Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```python
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
```

**Callback functions**

`ctypes` allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The `CFUNCTYPE()` factory function creates types for callback functions using the cdecl calling convention. On Windows, the `WINFUNCTYPE()` factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.
I will present an example here which uses the standard C library's `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```python
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```python
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
... >>> cmp_func = CMPFUNC(py_cmp_func)
```

To get started, here is a simple callback that shows the values it gets passed:

```python
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
... >>> cmp_func = CMPFUNC(py_cmp_func)
```
The result:

```python
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
```
```
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
```  

Now we can actually compare the two items and return a useful result:

```python
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
```
```
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
```  

As we can easily check, our array is sorted now:

```python
>>> for i in ia: print(i, end=",")
... 1 5 7 33 99
```  

The function factories can be used as decorator factories, so we may as well write:
```python
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
```

`Note:` Make sure you keep references to `CFUNCTYPE()` objects as long as they are used from C code. `ctypes` doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python’s control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with `threading.local` will *not* survive across different callbacks, even when those calls are made from the same C thread.

### Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.
**ctypes** can access values like this with the **in_dll()** class methods of the type. **pythonapi** is a predefined symbol giving access to the Python C api:

```python
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the **PyImport_FrozenModules** pointer exported by Python.

Quoting the docs for that value:

This pointer is initialized to point to an array of **struct _frozen** records, terminated by one whose members are all **NULL** or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with **ctypes**:

```python
>>> from ctypes import *

>>> class struct_frozen(Structure): ...
...   _fields_ = [("name", c_char_p), ...
...   ("code", POINTER(c_ubyte)), ...
...   ("size", c_int)]
```
We have defined the `struct _frozen` data type, so we can get the pointer to the table:

```python
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_

Since `table` is a pointer to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```python
>>> for item in table:
...    if item.name is None:
...        break
...    print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.
Surprises

There are some edges in ctypes where you might expect something other than what actually happens.

Consider the following example:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
```
Note that temp0 and temp1 are objects still using the internal buffer of the rc object above. So executing rc.a = temp0 copies the buffer contents of temp0 into rc's buffer. This, in turn, changes the contents of temp1. So, the last assignment rc.b = temp1, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't copy the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```python
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
```

Why is it printing False? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the contents of the object is stored. Accessing the contents again constructs a new Python object each time!

**Variable-sized data types**

ctypes provides some support for variable-sized arrays and structures.
The `resize()` function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a `ValueError` is raised if this is tried:

```python
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
  ... 
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>> 
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```python
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
  ... 
IndexError: invalid index
>>> 
```

Another way to use variable-sized data types with ctypes is to use the dynamic nature of Python, and (re-)define the data type after the
required size is already known, on a case by case basis.
Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.find_library(name)`

Try to find a library and return a pathname. `name` is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

*Changed in version 3.6:* On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.
Here are some examples:

```python
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
```  

On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```python
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
```  

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.  

If wrapping a shared library with `ctypes`, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to
locate the library at runtime.

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

```python
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
```

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return int.

```python
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `OSError` is automatically raised.

*Changed in version 3.3:* `WindowsError` used to be raised.

```python
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return int by default.

On Windows CE only the standard calling convention is used, for
convenience the WinDLL and OleDLL use the standard calling convention on this platform.

The Python global interpreter lock is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)

Instances of this class behave like CDLL instances, except that the Python GIL is not released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the handle named parameter, otherwise the underlying platforms dlopen or LoadLibrary function is used to load the library into the process, and to get a handle to it.

The mode parameter can be used to specify how the library is loaded. For details, consult the dlopen(3) manpage. On Windows, mode is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The use_errno parameter, when set to true, enables a ctypes mechanism that allows accessing the system errno error number in a safe way. ctypes maintains a thread-local copy of the systems errno variable; if you call foreign functions created with use_errno=True then the errno value before the function call is swapped with the ctypes private copy, the same happens immediately
after the function call.

The function `ctypes.get_errno()` returns the value of the ctypes private copy, and the function `ctypes.set_errno()` changes the ctypes private copy to a new value and returns the former value.

The `use_last_error` parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the `GetLastError()` and `SetLastError()` Windows API functions; `ctypes.get_last_error()` and `ctypes.set_last_error()` are used to request and change the ctypes private copy of the windows error code.

`ctypes.RTLD_GLOBAL`

Flag to use as `mode` parameter. On platforms where this flag is not available, it is defined as the integer zero.

`ctypes.RTLD_LOCAL`

Flag to use as `mode` parameter. On platforms where this is not available, it is the same as `RTLD_GLOBAL`.

`ctypes.DEFAULT_MODE`

The default mode which is used to load shared libraries. On OSX 10.3, this is `RTLD_GLOBAL`, otherwise it is the same as `RTLD_LOCAL`.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc["time"] == libc["time"]
False

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

**PyDLL._handle**

The system handle used to access the library.

**PyDLL._name**

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

```python
class ctypes.LibraryLoader(dlltype):
    Class which loads shared libraries. `dlltype` should be one of the CDLL, PyDLL, WinDLL, or OleDLL types.

    __getattr__() has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

    LoadLibrary(name)
    Load a shared library into the process and return it. This method always returns a new instance of the library.
```
These prefabricated library loaders are available:

c{}t{}y{}p{}e{}s.\texttt{cdll}

    Creates \texttt{CDLL} instances.

c{}t{}y{}p{}e{}s.\texttt{windll}

    Windows only: Creates \texttt{WinDLL} instances.

c{}t{}y{}p{}e{}s.\texttt{oledll}

    Windows only: Creates \texttt{OleDLL} instances.

c{}t{}y{}p{}e{}s.\texttt{pydll}

    Creates \texttt{PyDLL} instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

c{}t{}y{}p{}e{}s.\texttt{pythonapi}

    An instance of \texttt{PyDLL} that exposes Python C API functions as attributes. Note that all these functions are assumed to return C \texttt{int}, which is of course not always the truth, so you have to assign the correct \texttt{restype} attribute to use these functions.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any c{}t{}y{}p{}e{}s data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

\texttt{class c{}t{}y{}p{}e{}s._FuncPtr}
Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

**restype**

Assign a ctypes type to specify the result type of the foreign function. Use None for void, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C int, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as restype and assign a callable to the errcheck attribute.

**argtypes**

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the stdcall calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the from_param() class method of the items in the argtypes tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a c_char_p item in the argtypes tuple will convert
a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a from_param() method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

**errcheck**

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

```
callable(result, func, arguments)
```

- `result` is what the foreign function returns, as specified by the restype attribute.

- `func` is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

- `arguments` is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

**exception** ctypes.** ArgumentError**

This exception is raised when a foreign function call cannot convert one of the passed arguments.
Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the @wrapper syntax. See Callback functions for examples.

ctypes.**CFUNCTYPE**(restype, *argtypes, use_errno=False, use_last_error=False)

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If use_errno is set to true, the ctypes private copy of the system errno variable is exchanged with the real errno value before and after the call; use_last_error does the same for the Windows error code.

ctypes.**WINFUNCTYPE**(restype, *argtypes, use_errno=False, use_last_error=False)

Windows only: The returned function prototype creates functions that use the stdcall calling convention, except on Windows CE where WINFUNCTYPE() is the same as CFUNCTYPE(). The function will release the GIL during the call. use_errno and use_last_error have the same meaning as above.

ctypes.**PYFUNCTYPE**(restype, *argtypes)

The returned function prototype creates functions that use the Python calling convention. The function will not release the GIL
during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

**prototype(address)**
- Returns a foreign function at the specified address which must be an integer.

**prototype(callable)**
- Create a C callable function (a callback function) from a Python **callable**.

**prototype(func_spec[, paramflags])**
- Returns a foreign function exported by a shared library. **func_spec** must be a 2-tuple (**name_or_ordinal, library**). The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

**prototype(vtbl_index, name[, paramflags[, iid]])**
- Returns a foreign function that will call a COM method. **vtbl_index** is the index into the virtual function table, a small non-negative integer. **name** is name of the COM method. **iid** is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the **argtypes** tuple.

The optional **paramflags** parameter creates foreign function
wrappers with much more functionality than the features described above.

`paramflags` must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

- 1: Specifies an input parameter to the function.
- 2: Output parameter. The foreign function fills in a value.
- 4: Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```c
WINUSERAPI int WINAPI MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```
Here is the wrapping with **ctypes**:

```python
from ctypes import c_int, WINFUNCTYPE, windll
from ctypes.wintypes import HWND, LPCWSTR, UINT

prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, LP
paramflags = (1, "hwnd", 0), (1, "text", "Hi"),
MessageBox = prototype(("MessageBoxW", windll.us

The **MessageBox** foreign function can now be called in these ways:

```python
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 **GetWindowRect** function retrieves the dimensions of a specified window by copying them into **RECT** structure that the caller has to supply. Here is the C declaration:

```c
WINUSERAPI BOOL WINAPI GetWindowRect(  
    HWND hWnd,  
    LPRECT lpRect);
```

Here is the wrapping with **ctypes**:

```python
from ctypes import POINTER, WINFUNCTYPE, windll,  
from ctypes.wintypes import BOOL, HWND, RECT

prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT)
paramflags = (1, "hwnd"), (2, "lprect")
GetWindowRect = prototype(("GetWindowRect", windi
```
Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the errcheck protocol to do further output processing and error checking. The win32 GetWindowRect api function returns a BOOL to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```python
>>> def errcheck(result, func, args):
    ...
    if not result:
        ...
        raise WinError()
    ...
    return args
    ...
>>> GetWindowRect.errcheck = errcheck
>>>```

If the errcheck function returns the argument tuple it receives unchanged, ctypes continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a RECT instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```python
>>> def errcheck(result, func, args):
    ...
    if not result:
        ...
        raise WinError()
    ...
    rc = args[1]
    ...
    return rc.left, rc.top, rc.bottom, rc.right
    ...
>>> GetWindowRect.errcheck = errcheck
>>>```
Utility functions

ctypes.**addressof**(obj)

Returns the address of the memory buffer as integer. obj must be an instance of a ctypes type.

ctypes.**alignment**(*obj_or_type*)

Returns the alignment requirements of a ctypes type. obj_or_type must be a ctypes type or instance.

ctypes.**byref**(obj[, offset])

Returns a light-weight pointer to obj, which must be an instance of a ctypes type. offset defaults to zero, and must be an integer that will be added to the internal pointer value.

```
byref(obj, offset) corresponds to this C code:

(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to **pointer**(obj), but the construction is a lot faster.

ctypes.**cast**(obj, type)

This function is similar to the cast operator in C. It returns a new instance of type which points to the same memory block as obj. type must be a pointer type, and obj must be an object that can be interpreted as a pointer.

ctypes.**create_string_buffer**(init_or_size[, size=None])

This function creates a mutable character buffer. The returned object is a ctypes array of **c_char**.
*init_or_size* must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the bytes should not be used.

**ctypes.create_unicode_buffer**(*init_or_size, size=None*)

This function creates a mutable unicode character buffer. The returned object is a ctypes array of `c_wchar`.

*init_or_size* must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

**ctypes.DllCanUnloadNow()**

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

**ctypes.DllGetClassObject()**

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the _ctypes extension dll exports.
ctypes.util.\texttt{find\_library}(name)

Try to find a library and return a pathname. \textit{name} is the library name without any prefix like \texttt{lib}, suffix like \texttt{.so}, \texttt{.dylib} or version number (this is the form used for the posix linker option -l). If no library can be found, returns \texttt{None}.

The exact functionality is system dependent.

c\texttt{types.util.\texttt{find\_msvcrt}}()

Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, \texttt{None} is returned.

If you need to free memory, for example, allocated by an extension module with a call to the \texttt{free(void *)}, it is important that you use the function in the same library that allocated the memory.

c\texttt{types.\texttt{FormatError}([\textit{code}])}

Windows only: Returns a textual description of the error code \textit{code}. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

c\texttt{types.\texttt{GetLastError}}()

Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows \texttt{GetLastError()} function directly, it does not return the ctypes-private copy of the error code.

c\texttt{types.\texttt{get\_errno}()}

Returns the current value of the ctypes-private copy of the system \texttt{errno} variable in the calling thread.

c\texttt{types.\texttt{get\_last\_error}()}

Windows only: returns the current value of the ctypes-private copy of the system LastError variable in the calling thread.

**ctypes.memmove(dst, src, count)**

Same as the standard C memmove library function: copies `count` bytes from `src` to `dst`. `dst` and `src` must be integers or ctypes instances that can be converted to pointers.

**ctypes.memset(dst, c, count)**

Same as the standard C memset library function: fills the memory block at address `dst` with `count` bytes of value `c`. `dst` must be an integer specifying an address, or a ctypes instance.

**ctypes.POINTER(type)**

This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. `type` must be a ctypes type.

**ctypes.pointer(obj)**

This function creates a new pointer instance, pointing to `obj`. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

**ctypes.resize(obj, size)**

This function resizes the internal memory buffer of `obj`, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

**ctypes.set_errno(value)**

Set the current value of the ctypes-private copy of the system
errno variable in the calling thread to value and return the previous value.

```python
c ctypes.set_last_error(value)
```
Windows only: set the current value of the ctypes-private copy of the system LastError variable in the calling thread to value and return the previous value.

```python
c ctypes.sizeof(obj_or_type)
```
Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

```python
c ctypes.string_at(address, size=-1)
```
This function returns the C string starting at memory address address as a bytes object. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

```python
c ctypes.WinError(code=None, descr=None)
```
Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of OSError. If code is not specified, GetLastError is called to determine the error code. If descr is not specified, FormatError() is called to get a textual description of the error.

*Changed in version 3.3:* An instance of WindowsError used to be created.

```python
c ctypes.wstring_at(address, size=-1)
```
This function returns the wide character string starting at memory address address as a string. If size is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.
Data types

class ctypes._CData

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the `addressof()` helper function. Another instance variable is exposed as `_objects`; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the `metaclass`):

`from_buffer(source[, offset])`
This method returns a ctypes instance that shares the buffer of the `source` object. The `source` object must support the writeable buffer interface. The optional `offset` parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

`from_buffer_copy(source[, offset])`
This method creates a ctypes instance, copying the buffer from the `source` object buffer which must be readable. The optional `offset` parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

`from_address(address)`
This method returns a ctypes type instance using the memory specified by `address` which must be an integer.
**from_param(obj)**

This method adapts obj to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's argtypes tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns obj if that is an instance of the type. Some types accept other objects as well.

**in_dll(library, name)**

This method returns a ctypes type instance exported by a shared library. name is the name of the symbol that exports the data, library is the loaded shared library.

Common instance variables of ctypes data types:

**_b_base_**

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The _b_base_ read-only member is the root ctypes object that owns the memory block.

**_b_needsfree_**

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

**_objects**

This member is either None or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.
Fundamental data types

class ctypes._SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. _SimpleCData is a subclass of _CData, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the value attribute is retrieved from a ctypes instance, usually a new object is returned each time. ctypes does not implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a restype of c_char_p, you will always receive a Python bytes object, not a c_char_p instance.

Subclasses of fundamental data types do not inherit this behavior. So, if a foreign functions restype is a subclass of c_void_p, you will
receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

```python
class ctypes.c_byte
    Represents the C signed char datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char
    Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_char_p
    Represents the C char * datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.c_double
    Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble
    Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to `c_double`.

class ctypes.c_float
    Represents the C float datatype. The constructor accepts an
optional float initializer.

```python
class ctypes.c_int
```
Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias to `c_long`.

```python
class ctypes.c_int8
```
Represents the C 8-bit signed int datatype. Usually an alias for `c_byte`.

```python
class ctypes.c_int16
```
Represents the C 16-bit signed int datatype. Usually an alias for `c_short`.

```python
class ctypes.c_int32
```
Represents the C 32-bit signed int datatype. Usually an alias for `c_int`.

```python
class ctypes.c_int64
```
Represents the C 64-bit signed int datatype. Usually an alias for `c_longlong`.

```python
class ctypes.c_long
```
Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

```python
class ctypes.c_longlong
```
Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

```python
class ctypes.c_short
```
Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

```python
class ctypes.c_size_t
    Represents the C size_t datatype.
```

```python
class ctypes.c_ssize_t
    Represents the C ssize_t datatype.
```

*New in version 3.2.*

```python
class ctypes.c_ubyte
    Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.
```

```python
class ctypes.c_uint
    Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for `c_ulong`.
```

```python
class ctypes.c_uint8
    Represents the C 8-bit unsigned int datatype. Usually an alias for `c_ubyte`.
```

```python
class ctypes.c_uint16
    Represents the C 16-bit unsigned int datatype. Usually an alias for `c_ushort`.
```

```python
class ctypes.c_uint32
    Represents the C 32-bit unsigned int datatype. Usually an alias for `c_uint`.
```
class ctypes.c_uint64
   Represents the C 64-bit unsigned int datatype. Usually an alias for c_ulonglong.

class ctypes.c_ulong
   Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ulonglong
   Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ushort
   Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_void_p
   Represents the C void * type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.c_wchar
   Represents the C wchar_t datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_wchar_p
   Represents the C wchar_t * datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.
class ctypes.c_bool
Represent the C bool datatype (more accurately, _Bool from C99). Its value can be True or False, and the constructor accepts any object that has a truth value.

class ctypes.HRESULT
Windows only: Represents a HRESULT value, which contains success or error information for a function or method call.

class ctypes.py_object
Represents the C PyObject * datatype. Calling this without an argument creates a NULL PyObject * pointer.

The ctypes.wintypes module provides quite some other Windows specific data types, for example HWND, WPARAM, or DWORD. Some useful structures like MSG or RECT are also defined.

Structured data types

class ctypes.Union(*args, **kw)
Abstract base class for unions in native byte order.

class ctypes.BigEndianStructure(*args, **kw)
Abstract base class for structures in big endian byte order.

class ctypes.LittleEndianStructure(*args, **kw)
Abstract base class for structures in little endian byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.Structure(*args, **kw)
Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a `_fields_` class variable. `ctypes` will create descriptors which allow reading and writing the fields by direct attribute accesses. These are the

```python
_fields_
```

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like `c_int`, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the `_fields_` class variable after the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```python
class List(Structure):
    pass
List._fields_ = ["pnext", POINTER(List)], ...
```

The `_fields_` class variable must, however, be defined before
the type is first used (an instance is created, \texttt{sizeof()} is called on it, and so on). Later assignments to the \_fields\_ class variable will raise an AttributeError.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the \_fields\_ defined in the sub-subclass, if any.

\texttt{\_pack\_}

An optional small integer that allows overriding the alignment of structure fields in the instance. \_pack\_ must already be defined when \_fields\_ is assigned, otherwise it will have no effect.

\texttt{\_anonymous\_}

An optional sequence that lists the names of unnamed (anonymous) fields. \_anonymous\_ must be already defined when \_fields\_ is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. \texttt{ctypes} will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```python
class \_U(Union):
    \_fields_ = ["lptdesc", \_U],
    \_anonymous_ = ("u",)
    \_fields_ = ["u", \_U],
```

```python
class TYPEDESC(Structure):
    \_anonymous_ = ("u",)
    \_fields_ = ["u", \_U],
```
The TYPEDESC structure describes a COM data type, the vt field specifies which one of the union fields is valid. Since the u field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. td.lptdesc and td.u.lptdesc are equivalent, but the former is faster since it does not need to create a temporary union instance:

```python
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate _fields_ variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in _fields_. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize _fields_ with the same name, or create new attributes for names not present in _fields_.

**Arrays and pointers**

```python
class ctypes.Array(*args)
```

Abstract base class for arrays.
The recommended way to create concrete array types is by multiplying any ctypes data type with a positive integer. Alternatively, you can subclass this type and define _length_ and _type_ class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is not itself an Array.

_length_
A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an IndexError. Will be returned by len().

_type_
Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class ctypes._Pointer
Private, abstract base class for pointers.

Concrete pointer types are created by calling POINTER() with the type that will be pointed to; this is done automatically by pointer().

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so len() will raise TypeError. Negative subscripts will read from the memory before the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

_type_
Specifies the type pointed to.

**contents**

Returns the object to which the pointer points. Assigning to this attribute changes the pointer to point to the assigned object.
Concurrent Execution

The modules described in this chapter provide support for concurrent execution of code. The appropriate choice of tool will depend on the task to be executed (CPU bound vs IO bound) and preferred style of development (event driven cooperative multitasking vs preemptive multitasking). Here's an overview:

- **threading — Thread-based parallelism**
  - Thread-Local Data
  - Thread Objects
  - Lock Objects
  - RLock Objects
  - Condition Objects
  - Semaphore Objects
    - Semaphore Example
  - Event Objects
  - Timer Objects
  - Barrier Objects
  - Using locks, conditions, and semaphores in the with statement

- **multiprocessing — Process-based parallelism**
  - Introduction
    - The Process class
    - Contexts and start methods
    - Exchanging objects between processes
    - Synchronization between processes
    - Sharing state between processes
    - Using a pool of workers
  - Reference
    - Process and exceptions
    - Pipes and Queues
• Miscellaneous
• Connection Objects
• Synchronization primitives
• Shared ctypes Objects
  • The multiprocessing.sharedctypes module
• Managers
  • Customized managers
  • Using a remote manager
• Proxy Objects
  • Cleanup
• Process Pools
• Listeners and Clients
  • Address Formats
• Authentication keys
• Logging
  • The multiprocessing.dummy module
  • Programming guidelines
    • All start methods
    • The spawn and forserver start methods
  • Examples
• The concurrent package
• concurrent.futures — Launching parallel tasks
  • Executor Objects
  • ThreadPoolExecutor
    • ThreadPoolExecutor Example
  • ProcessPoolExecutor
    • ProcessPoolExecutor Example
  • Future Objects
  • Module Functions
  • Exception classes
• subprocess — Subprocess management
  • Using the subprocess Module
    • Frequently Used Arguments
    • Popen Constructor
- Exceptions
  - Security Considerations
  - Popen Objects
  - Windows Popen Helpers
    - Windows Constants
  - Older high-level API
  - Replacing Older Functions with the subprocess Module
    - Replacing /bin/sh shell backquote
    - Replacing shell pipeline
    - Replacing os.system()
    - Replacing the os.spawn family
    - Replacing os.popen(), os.popen2(), os.popen3()
    - Replacing functions from the popen2 module
  - Legacy Shell Invocation Functions
  - Notes
    - Converting an argument sequence to a string on Windows
  - sched — Event scheduler
    - Scheduler Objects
  - queue — A synchronized queue class
    - Queue Objects
    - SimpleQueue Objects

The following are support modules for some of the above services:

- _thread — Low-level threading API
- _dummy_thread — Drop-in replacement for the _thread module
- dummy_threading — Drop-in replacement for the threading module
**threading** — Thread-based parallelism

**Source code:** Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level `_thread` module. See also the `queue` module.

*Changed in version 3.7:* This module used to be optional, it is now always available.

**Note:** While they are not listed below, the camelCase names used for some methods and functions in this module in the Python 2.x series are still supported by this module.

This module defines the following functions:

```python
threading.active_count()
```

Return the number of `Thread` objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.

```python
threading.current_thread()
```

Return the current `Thread` object, corresponding to the caller's thread of control. If the caller’s thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

```python
threading.get_ident()
```

Return the ‘thread identifier’ of the current thread. This is a nonzero
integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

*New in version 3.3.*

```
threading.enumerate()
```
Return a list of all `Thread` objects currently alive. The list includes daemonic threads, dummy thread objects created by `current_thread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

```
threading.main_thread()
```
Return the main `Thread` object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

*New in version 3.4.*

```
threading.settrace(func)
```
Set a trace function for all threads started from the `threading` module. The `func` will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

```
threading.setprofile(func)
```
Set a profile function for all threads started from the `threading` module. The `func` will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

```
threading.stack_size([size])
```
Return the thread stack size used when creating new threads. The optional `size` argument specifies the stack size to be used for
subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If size is not specified, 0 is used. If changing the thread stack size is unsupported, a RuntimeError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

**Availability:** Windows, systems with POSIX threads.

This module also defines the following constant:

```
threading.TIMEOUT_MAX
```

The maximum value allowed for the timeout parameter of blocking functions (Lock.acquire(), RLock.acquire(), Condition.wait(), etc.). Specifying a timeout greater than this value will raise an OverflowError.

*New in version 3.2.*

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python’s
Thread class supports a subset of the behavior of Java’s Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java’s Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.
Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of `local` (or a subclass) and store attributes on it:

```python
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

`class threading.local`

A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the `_threading_local` module.
Thread Objects

The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the `__init__()` and `run()` methods of this class.

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

Once the thread’s activity is started, the thread is considered ‘alive’. It stops being alive when its `run()` method terminates – either normally, or by raising an unhandled exception. The `is_alive()` method tests whether the thread is alive.

Other threads can call a thread’s `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the `name` attribute.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property or the `daemon` constructor argument.

**Note:** Daemon threads are abruptly stopped at shutdown. Their
resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an Event.

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be join()ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ()

kwargs is a dictionary of keyword arguments for the target
If not None, daemon explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

*Changed in version 3.3:* Added the daemon argument.

**start()**

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

**run()**

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

**join(timeout=None)**

Wait until the thread terminates. This blocks the calling thread
until the thread whose `join()` method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the `timeout` argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join()`ed many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raise the same exception.

**name**

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**getName()**
**setName()**

Old getter/setter API for `name`; use it directly as a property instead.

**ident**

The ‘thread identifier’ of this thread or `None` if the thread has
not been started. This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

**is_alive()**

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

**daemon**

A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

**isDaemon()**
**setDaemon()**

Old getter/setter API for daemon; use it directly as a property instead.

**CPython implementation detail:** In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to
use **multiprocessing** or **concurrent.futures.ProcessPoolExecutor**. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.
Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the _thread extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

Locks also support the context management protocol.

When more than one thread is blocked in acquire() waiting for the state to turn to unlocked, only one thread proceeds when a release() call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.Lock

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.
Note that Lock is actually a factory function which returns an instance of the most efficient version of the concrete Lock class that is supported by the platform.

**acquire**(blocking=True, timeout=-1)

Acquire a lock, blocking or non-blocking.

When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.

When invoked with the blocking argument set to False, do not block. If a call with blocking set to True would block, return False immediately; otherwise, set the lock to locked and return True.

When invoked with the floating-point timeout argument set to a positive value, block for at most the number of seconds specified by timeout and as long as the lock cannot be acquired. A timeout argument of -1 specifies an unbounded wait. It is forbidden to specify a timeout when blocking is false.

The return value is True if the lock is acquired successfully, False if not (for example if the timeout expired).

*Changed in version 3.2:* The timeout parameter is new.

*Changed in version 3.2:* Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

**release**()

Release a lock. This can be called from any thread, not only the
thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

There is no return value.
RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its `acquire()` method; this returns once the thread owns the lock. To unlock the lock, a thread calls its `release()` method. `acquire()/release()` call pairs may be nested; only the final `release()` (the `release()` of the outermost pair) resets the lock to unlocked and allows another thread blocked in `acquire()` to proceed.

Reentrant locks also support the context management protocol.

```python
class threading.RLock
    This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

    Note that RLock is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

    acquire(blocking=True, timeout=-1)
    Acquire a lock, blocking or non-blocking.
    When invoked without arguments: if this thread already owns
the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the `blocking` argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the `blocking` argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point `timeout` argument set to a positive value, block for at most the number of seconds specified by `timeout` and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

*Changed in version 3.2:* The `timeout` parameter is new.

**release()**

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A
**RuntimeError** is raised if this method is called when the lock is unlocked.

There is no return value.
Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don’t have to track it separately.

A condition variable obeys the context management protocol: using the `with` statement acquires the associated lock for the duration of the enclosed block. The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notify_all()` methods don’t release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are
interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```python
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application’s condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts:

```python
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding
one item to the buffer only needs to wake up one consumer thread.

class threading.Condition(lock=None)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the lock argument is given and not None, it must be a Lock or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.

*Changed in version 3.3:* changed from a factory function to a class.

acquire(*args)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(timeout=None)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

This method releases the underlying lock, and then blocks until it is awakened by a notify() or notify_all() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.
When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using its `release()` method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the `RLock` class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given `timeout` expired, in which case it is `False`.

*Changed in version 3.2:* Previously, the method always returned `None`.

`wait_for(predicate, timeout=None)`

Wait until a condition evaluates to true. `predicate` should be a callable which result will be interpreted as a boolean value. A `timeout` may be provided giving the maximum time to wait.

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```python
while not predicate():
    cv.wait()
```
Therefore, the same rules apply as with `wait()`: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

*New in version 3.2.*

**notify**(n=1)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most \( n \) of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly \( n \) threads, if at least \( n \) threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than \( n \) threads.

Note: an awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

**notify_all()**

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.
Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names $P()$ and $V()$ instead of $\text{acquire()}$ and $\text{release()}$).

A semaphore manages an internal counter which is decremented by each $\text{acquire()}$ call and incremented by each $\text{release()}$ call. The counter can never go below zero; when $\text{acquire()}$ finds that it is zero, it blocks, waiting until some other thread calls $\text{release()}$.

Semaphores also support the context management protocol.

```python
class threading.Semaphore(value=1)
```

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of $\text{release()}$ calls minus the number of $\text{acquire()}$ calls, plus an initial value. The $\text{acquire()}$ method blocks if necessary until it can return without making the counter negative. If not given, $value$ defaults to 1.

The optional argument gives the initial $value$ for the internal counter; it defaults to 1. If the $value$ given is less than 0, $ValueError$ is raised.

*Changed in version 3.3:* changed from a factory function to a class.

```python
acquire(blocking=True, timeout=None)
```

Acquire a semaphore.

When invoked without arguments:
- If the internal counter is larger than zero on entry, decrement it by one and return true immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return true. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with `blocking` set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a `timeout` other than `None`, it will block for at most `timeout` seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

*Changed in version 3.2*: The `timeout` parameter is new.

### `release()`

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

### `class threading.BoundedSemaphore(value=1)`

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn’t exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it’s a sign of a bug. If not given, `value` defaults to 1.

*Changed in version 3.3*: changed from a factory function to a class.
**Semaphore Example**

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```python
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore’s acquire and release methods when they need to connect to the server:

```python
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it’s acquired will go undetected.
Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

```python
class threading.Event
    Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.
```

*Changed in version 3.3:* changed from a factory function to a class.

`is_set()`
Return true if and only if the internal flag is true.

`set()`
Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

`clear()`
Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

`wait(timeout=None)`
Block until the internal flag is true. If the internal flag is true on
entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns true if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return `True` except if a timeout is given and the operation times out.

*Changed in version 3.1:* Previously, the method always returned `None`. 
Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the cancel() method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```python
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

```
class threading.Timer(interval, function, args=None, kwargs=None)

Create a timer that will run function with arguments args and keyword arguments kwargs, after interval seconds have passed. If args is None (the default) then an empty list will be used. If kwargs is None (the default) then an empty dict will be used.

Changed in version 3.3: changed from a factory function to a class.
```

```
cancel()

Stop the timer, and cancel the execution of the timer’s action. This will only work if the timer is still in its waiting stage.
```
Barrier Objects

New in version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the `wait()` method and will block until all of the threads have made their `wait()` calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```python
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.Barrier(parties, action=None, timeout=None)

Create a barrier object for `parties` number of threads. An `action`, when provided, is a callable to be called by one of the threads
when they are released. *timeout* is the default timeout value if none is specified for the `wait()` method.

### `wait(timeout=None)`

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a `timeout` is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to parties – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```python
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a thread is waiting.

### `reset()`

Return the barrier to the default, empty state. Any threads waiting on it will receive the `BrokenBarrierError` exception.

Note that using this function may can require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it
and create a new one.

**abort()**

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible `timeout` value to automatically guard against one of the threads going awry.

**parties**

The number of threads required to pass the barrier.

**n_waiting**

The number of threads currently waiting in the barrier.

**broken**

A boolean that is `True` if the barrier is in the broken state.

**exception** threading. `BrokenBarrierError`

This exception, a subclass of `RuntimeError`, is raised when the `Barrier` object is reset or broken.
Using locks, conditions, and semaphores in the with statement

All of the objects provided by this module that have acquire() and release() methods can be used as context managers for a `with` statement. The acquire() method will be called when the block is entered, and release() will be called when the block is exited. Hence, the following snippet:

```python
with some_lock:
    # do something...
```

is equivalent to:

```python
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, `Lock`, `RLock`, `Condition`, `Semaphore`, and `BoundedSemaphore` objects may be used as `with` statement context managers.
multiprocessing — Process-based parallelism

Source code: Lib/multiprocessing/
Introduction

**multiprocessing** is a package that supports spawning processes using an API similar to the **threading** module. The **multiprocessing** package offers both local and remote concurrency, effectively side-stepping the **Global Interpreter Lock** by using subprocesses instead of threads. Due to this, the **multiprocessing** module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The **multiprocessing** module also introduces APIs which do not have analogs in the **threading** module. A prime example of this is the **Pool** object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using **Pool**,

```python
from multiprocessing import Pool

def f(x):
    return x * x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```
The **Process** class

In **multiprocessing**, processes are spawned by creating a **Process** object and then calling its **start()** method. **Process** follows the API of **threading.Thread**. A trivial example of a multiprocess program is

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```python
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
```
For an explanation of why the `if __name__ == '__main__'` part is necessary, see Programming guidelines.

## Contexts and start methods

Depending on the platform, `multiprocessing` supports three ways to start a process. These *start methods* are

### spawn

The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process objects `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using `fork` or `forkserver`.

Available on Unix and Windows. The default on Windows.

### fork

The parent process uses `os.fork()` to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Available on Unix only. The default on Unix.

### forkserver

When the program starts and selects the `forkserver` start
method, a server process is started. From then on, whenever
a new process is needed, the parent process connects to the
server and requests that it fork a new process. The fork server
process is single threaded so it is safe for it to use
\texttt{os.fork()}. No unnecessary resources are inherited.

Available on Unix platforms which support passing file
descriptors over Unix pipes.

\textit{Changed in version 3.4: spawn} added on all unix platforms, and
\textit{forkserver} added for some unix platforms. Child processes no longer
inherit all of the parents inheritable handles on Windows.

On Unix using the \textit{spawn} or \textit{forkserver} start methods will also start a
\textit{semaphore tracker} process which tracks the unlinked named
semaphores created by processes of the program. When all
processes have exited the semaphore tracker unlinks any remaining
semaphores. Usually there should be none, but if a process was killed
by a signal there may be some “leaked” semaphores. (Unlinking the
named semaphores is a serious matter since the system allows only a
limited number, and they will not be automatically unlinked until the
next reboot.)

To select a start method you use the \texttt{set_start_method()} in the
\texttt{if \_\_name\_ == \'_\_main\_\_'} clause of the main module. For
example:

\begin{verbatim}
import multiprocessing as mp
def foo(q):
    q.put('hello')
if \_\_name\_ == \'_\_main\_\_':
    mp.set_start_method('spawn')
\end{verbatim}
```python
q = mp.Queue()
p = mp.Process(target=foo, args=(q,))
p.start()
print(q.get())
p.join()
```

`set_start_method()` should not be used more than once in the program.

Alternatively, you can use `get_context()` to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```python
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
p.start()
print(q.get())
p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the `fork` context cannot be passed to processes started using the `spawn` or `forkserver` start methods.

A library which wants to use a particular start method should probably use `get_context()` to avoid interfering with the choice of the library user.
Warning: The 'spawn' and 'forkserver' start methods cannot currently be used with "frozen" executables (i.e., binaries produced by packages like PyInstaller and cx_Freeze) on Unix. The 'fork' start method does work.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The Queue class is a near clone of queue.Queue. For example:

```python
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())  # prints '[42, None, 'hello']
    p.join()
```

Queues are thread and process safe.

Pipes

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())  # prints "[42, None, 'hello']"
    p.join()

The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv() methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the same end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from threading. For instance one can use a lock to ensure that only one process prints to standard output at a time:

from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
Without using the lock output from the different processes is liable to get all mixed up.

**Sharing state between processes**

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then *multiprocessing* provides a couple of ways of doing so.

**Shared memory**

Data can be stored in a shared memory map using *Value* or *Array*. For example, the following code

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
```
arr = Array('i', range(10))

p = Process(target=f, args=(num, arr))
p.start()
p.join()

print(num.value)
print(arr[:])

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The 'd' and 'i' arguments used when creating num and arr are typecodes of the kind used by the array module: 'd' indicates a double precision float and 'i' indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the multiprocessing.sharedctypes module which supports the creation of arbitrary ctypes objects allocated from shared memory.

**Server process**

A manager object returned by Manager() controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by Manager() will support types list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier, Queue, Value and Array. For example,
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)

will print

{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The Pool class represents a pool of worker processes. It has
methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```python
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x**x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))  # runs
        print(res.get(timeout=1))  # print

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ())  # runs
        print(res.get(timeout=1))  # print

        # launching multiple evaluations asynchronously
        multiple_results = [pool.apply_async(os.getpid, ()) for _ in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
```
try:
    print(res.get(timeout=1))
except TimeoutError:
    print("We lacked patience and got a multiprocessing.TimeoutError")

print("For the moment, the pool remains available")
# exiting the 'with'-block has stopped the pool
print("Now the pool is closed and no longer available")

Note that the methods of a pool should only ever be used by the process which created it.

---

**Note:** Functionality within this package requires that the ``__main__`` module be importable by the children. This is covered in [Programming guidelines](http://www.example.com/programming-guidelines) however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```python
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x**x
... >>> p.map(f, [1,2,3])
ProcessPoolWorker-1:
ProcessPoolWorker-2:
ProcessPoolWorker-3:
Traceback (most recent call last):
  File "", line 1, in 
    AttributeError: 'module' object has no attribute 'f'
  File "", line 1, in 
    AttributeError: 'module' object has no attribute 'f'
  File "", line 1, in 
    AttributeError: 'module' object has no attribute 'f'
```
(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)
Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

**Process and exceptions**

class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. `group` should always be `None`; it exists solely for compatibility with `threading.Thread`. `target` is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. `name` is the process name (see `name` for more details). `args` is the argument tuple for the target invocation. `kwargs` is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only `daemon` argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to `target`.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`)) before doing anything else to the process.

*Changed in version 3.3:* Added the `daemon` argument.
**run()**
Method representing the process's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

**start()**
Start the process's activity.

This must be called at most once per process object. It arranges for the object's `run()` method to be invoked in a separate process.

**join([timeout])**
If the optional argument `timeout` is `None` (the default), the method blocks until the process whose `join()` method is called terminates. If `timeout` is a positive number, it blocks at most `timeout` seconds. Note that the method returns `None` if its process terminates or if the method times out. Check the process's `exitcode` to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

**name**
The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be
given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form ‘Process-$N_1$:$N_2$: \ldots :$N_k$’ is constructed, where each $N_k$ is the $N$-th child of its parent.

**is_alive()**

Return whether the process is alive.

Roughly, a process object is alive from the moment the **start()** method returns until the child process terminates.

**daemon**

The process's daemon flag, a Boolean value. This must be set before **start()** is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the **threading.Thread** API, **Process** objects also support the following attributes and methods:

**pid**

Return the process ID. Before the process is spawned, this will
be None.

**exitcode**

The child’s exit code. This will be None if the process has not yet terminated. A negative value -N indicates that the child was terminated by signal \( N \).

**authkey**

The process’s authentication key (a byte string).

When `multiprocessing` is initialized the main process is assigned a random string using `os.urandom()`.

When a `Process` object is created, it will inherit the authentication key of its parent process, although this may be changed by setting `authkey` to another byte string.

See [Authentication keys](#).

**sentinel**

A numeric handle of a system object which will become “ready” when the process ends.

You can use this value if you want to wait on several events at once using `multiprocessing.connection.wait()`. Otherwise calling `join()` is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On Unix, this is a file descriptor usable with primitives from the `select` module.

*New in version 3.3.*
**terminate()**

terminate the process. On Unix this is done using the SIGTERM signal; on Windows TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will not be terminated – they will simply become orphaned.

**Warning:** If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

**kill()**

Same as terminate() but using the SIGKILL signal on Unix.

*New in version 3.7.*

**close()**

Close the Process object, releasing all resources associated with it. ValueError is raised if the underlying process is still running. Once close() returns successfully, most other methods and attributes of the Process object will raise ValueError.

*New in version 3.7.*

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only be called by the process that
created the process object.

Example usage of some of the methods of `Process`:

```python
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM]> False
>>> p.exitcode == -signal.SIGTERM
True
```

`exception` `multiprocessing.ProcessError` 
The base class of all `multiprocessing` exceptions.

`exception` `multiprocessing.BufferTooShort` 
Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of `BufferTooShort` then `e.args[0]` will give the message as a byte string.

`exception` `multiprocessing.AuthenticationError` 
Raised when there is an authentication error.

`exception` `multiprocessing.TimeoutError` 
Raised by methods with a timeout when the timeout expires.
Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use `Pipe()` (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The `Queue`, `SimpleQueue` and `JoinableQueue` types are multi-producer, multi-consumer **FIFO** queues modelled on the `queue.Queue` class in the standard library. They differ in that `Queue` lacks the `task_done()` and `join()` methods introduced into Python 2.5's `queue.Queue` class.

If you use `JoinableQueue` then you **must** call `JoinableQueue.task_done()` for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see Managers.

**Note:** `multiprocessing` uses the usual `queue.Empty` and `queue.Full` exceptions to signal a timeout. They are not available in the `multiprocessing` namespace so you need to import them from `queue`.

**Note:** When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but
should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a manager.

1. After putting an object on an empty queue there may be an infinitesimal delay before the queue’s empty() method returns False and get_nowait() can return without raising queue.Empty.

2. If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

**Warning:** If a process is killed using Process.terminate() or os.kill() while it is trying to use a Queue, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

**Warning:** As mentioned above, if a child process has put items on a queue (and it has not used JoinableQueue.cancel_join_thread), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See Programming guidelines.

For an example of the usage of queues for interprocess
communication see Examples.

multiprocessing.**Pipe**([**duplex**])

Returns a pair \((\text{conn1}, \text{conn2})\) of **Connection** objects representing the ends of a pipe.

If **duplex** is **True** (the default) then the pipe is bidirectional. If **duplex** is **False** then the pipe is unidirectional: \(\text{conn1}\) can only be used for receiving messages and \(\text{conn2}\) can only be used for sending messages.

class multiprocessing.**Queue**([**maxsize**])

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual **queue.Empty** and **queue.Full** exceptions from the standard library’s **queue** module are raised to signal timeouts.

**Queue** implements all the methods of **queue.Queue** except for **task_done()** and **join()**.

**qsize()**

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise **NotImplementedError** on Unix platforms like Mac OS X where **sem_getvalue()** is not implemented.

**empty()**
Return True if the queue is empty, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

**full()**
Return True if the queue is full, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

**put(obj[, block[, timeout]])**
Put obj into the queue. If the optional argument block is True (the default) and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the queue.Full exception if no free slot was available within that time. Otherwise (block is False), put an item on the queue if a free slot is immediately available, else raise the queue.Full exception (timeout is ignored in that case).

**put_nowait(obj)**
Equivalent to put(obj, False).

**get([block[, timeout]])**
Remove and return an item from the queue. If optional args block is True (the default) and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the queue.Empty exception if no item was available within that time. Otherwise (block is False), return an item if one is immediately available, else raise the queue.Empty exception (timeout is ignored in that case).

**get_nowait()**
Equivalent to `get(False)`.

`multiprocessing.Queue` has a few additional methods not found in `queue.Queue`. These methods are usually unnecessary for most code:

**close()**

Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

**join_thread()**

Join the background thread. This can only be used after `close()` has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue’s background thread. The process can call `cancel_join_thread()` to make `join_thread()` do nothing.

**cancel_join_thread()**

Prevent `join_thread()` from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see `join_thread()`.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to
exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

**Note:** This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a `Queue` will result in an `ImportError`. See bpo-3770 for additional information. The same holds true for any of the specialized queue types listed below.

```python
class multiprocessing.SimpleQueue

It is a simplified `Queue` type, very close to a locked `Pipe`.

**empty()**
Return `True` if the queue is empty, `False` otherwise.

**get()**
Remove and return an item from the queue.

**put(item)**
Put `item` into the queue.
```

class multiprocessing.JoinableQueue([maxsize])

`JoinableQueue`, a `Queue` subclass, is a queue which additionally has `task_done()` and `join()` methods.

**task_done()**
Indicate that a formerly enqueued task is complete. Used by queue consumers. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.
If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

**join()**
Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

**Miscellaneous**

`multiprocessing.active_children()`
Return list of all live children of the current process.

Calling this has the side effect of “joining” any processes which have already finished.

`multiprocessing.cpu_count()`
Return the number of CPUs in the system.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with `len(os.sched_getaffinity(0))`.

May raise `NotImplementedError`. 
See also: `os.cpu_count()`

`multiprocessing.current_process()`

Return the `Process` object corresponding to the current process.

An analogue of `threading.current_thread()`.

`multiprocessing.freeze_support()`

Add support for when a program which uses `multiprocessing` has been frozen to produce a Windows executable. (Has been tested with `py2exe`, `PylInstaller` and `cx_Freeze`.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```python
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise `RuntimeError`.

Calling `freeze_support()` has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then `freeze_support()` has no effect.
multiprocessing.get_all_start_methods()  
Returns a list of the supported start methods, the first of which is the default. The possible start methods are 'fork', 'spawn' and 'forkserver'. On Windows only 'spawn' is available. On Unix 'fork' and 'spawn' are always supported, with 'fork' being the default.

New in version 3.4.

multiprocessing.get_context(method=None)  
Return a context object which has the same attributes as the multiprocessing module.

If method is None then the default context is returned. Otherwise method should be 'fork', 'spawn', 'forkserver'. ValueError is raised if the specified start method is not available.

New in version 3.4.

multiprocessing.get_start_method(allow_none=False)  
Return the name of start method used for starting processes.

If the start method has not been fixed and allow_none is false, then the start method is fixed to the default and the name is returned. If the start method has not been fixed and allow_none is true then None is returned.

The return value can be 'fork', 'spawn', 'forkserver' or None. 'fork' is the default on Unix, while 'spawn' is the default on Windows.

New in version 3.4.
multiprocessing.set_executable()

Sets the path of the Python interpreter to use when starting a child process. (By default sys.executable is used). Embedders will probably need to do some thing like

```python
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

*Changed in version 3.4:* Now supported on Unix when the 'spawn' start method is used.

multiprocessing.set_start_method(method)

Set the method which should be used to start child processes. method can be 'fork', 'spawn' or 'forkserver'.

Note that this should be called at most once, and it should be protected inside the `if __name__ == '__main__'` clause of the main module.

*New in version 3.4.*

**Note:** multiprocessing contains no analogues of threading.active_count(), threading.enumerate(), threading.settrace(), threading.setprofile(), threading.Timer, or threading.local.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.
Connection objects are usually created using `Pipe` – see also Listeners and Clients.

```python
class multiprocessing.connection.Connection

    send(obj)
    Send an object to the other end of the connection which should be read using `recv()`.

    The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a `ValueError` exception.

    recv()
    Return an object sent from the other end of the connection using `send()`. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

    fileno()
    Return the file descriptor or handle used by the connection.

    close()
    Close the connection.

    This is called automatically when the connection is garbage collected.

    poll([timeout])
    Return whether there is any data available to be read.

    If `timeout` is not specified then it will return immediately. If `timeout` is a number then this specifies the maximum time in seconds to block. If `timeout` is `None` then an infinite timeout is
used.

Note that multiple connection objects may be polled at once by using `multiprocessing.connection.wait()`.

`send_bytes(buffer[, offset[, size]])`

Send byte data from a bytes-like object as a complete message.

If `offset` is given then data is read from that position in `buffer`. If `size` is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a `ValueError` exception.

`recv_bytes([maxlength])`

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end has closed.

If `maxlength` is specified and the message is longer than `maxlength` then `OSError` is raised and the connection will no longer be readable.

*Changed in version 3.3:* This function used to raise `IOError`, which is now an alias of `OSError`.

`recv_bytes_into(buffer[, offset])`

Read into `buffer` a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end
was closed.

_buffer_ must be a writable _bytes-like object_. If _offset_ is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of _buffer_ (in bytes).

If the buffer is too short then a BufferTooShort exception is raised and the complete message is available as _e.args[0]_ where _e_ is the exception instance.

*Changed in version 3.3:* Connection objects themselves can now be transferred between processes using _Connection.send()_ and _Connection.recv()_.

*New in version 3.3:* Connection objects now support the context management protocol – see _Context Manager Types_. __enter__() returns the connection object, and __exit__() calls close().

For example:

```python
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
```
```python
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

**Warning:** The `Connection.recv()` method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using `Pipe()` you should only use the `recv()` and `send()` methods after performing some sort of authentication. See [Authentication keys](#).

**Warning:** If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

### Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for [threading](#) module.

Note that one can also create synchronization primitives by using a manager object – see [Managers](#).

```python
class multiprocessing.Barrier(parties[, action[, timeout]])
```

A barrier object: a clone of `threading.Barrier`.

*New in version 3.3.*
**class multiprocessing.** **BoundedSemaphore**(\(\text{[value]}\))

A bounded semaphore object: a close analog of **threading.BoundedSemaphore**.

A solitary difference from its close analog exists: its **acquire** method's first argument is named *block*, as is consistent with **Lock.acquire()**.

**Note:** On Mac OS X, this is indistinguishable from **Semaphore** because *sem_getvalue()* is not implemented on that platform.

**class multiprocessing.** **Condition**(\(\text{[lock]}\))

A condition variable: an alias for **threading.Condition**.

If *lock* is specified then it should be a **Lock** or **RLock** object from **multiprocessing**.

*Changed in version 3.3:* The **wait_for()** method was added.

**class multiprocessing.** **Event**

A clone of **threading.Event**.

**class multiprocessing.** **Lock**

A non-recursive lock object: a close analog of **threading.Lock**.

Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of **threading.Lock** as it applies to threads are replicated here in **multiprocessing.Lock** as it applies to either processes or threads, except as noted.

Note that **Lock** is actually a factory function which returns an
instance of `multiprocessing.synchronize.Lock` initialized with a default context.

`Lock` supports the `context manager` protocol and thus may be used in `with` statements.

`acquire(block=True, timeout=None)`

Acquire a lock, blocking or non-blocking.

With the `block` argument set to `True` (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return `True`. Note that the name of this first argument differs from that in `threading.Lock.acquire()`.

With the `block` argument set to `False`, the method call does not block. If the lock is currently in a locked state, return `False`; otherwise set the lock to a locked state and return `True`.

When invoked with a positive, floating-point value for `timeout`, block for at most the number of seconds specified by `timeout` as long as the lock can not be acquired. Invocations with a negative value for `timeout` are equivalent to a `timeout` of zero. Invocations with a `timeout` value of `None` (the default) set the timeout period to infinite. Note that the treatment of negative or `None` values for `timeout` differs from the implemented behavior in `threading.Lock.acquire()`. The `timeout` argument has no practical implications if the `block` argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed.

`release()`

Release a lock. This can be called from any process or thread,
not only the process or thread which originally acquired the lock.

Behavior is the same as in `threading.Lock.release()` except that when invoked on an unlocked lock, a `ValueError` is raised.

```python
class multiprocessing.RLock
```

A recursive lock object: a close analog of `threading.RLock`. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that `RLock` is actually a factory function which returns an instance of `multiprocessing.synchronize.RLock` initialized with a default context.

`RLock` supports the context manager protocol and thus may be used in `with` statements.

```python
acquire(block=True, timeout=None)
```

Acquire a lock, blocking or non-blocking.

When invoked with the `block` argument set to `True`, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of `True`. Note that there are several differences in this first argument's behavior compared to the implementation
of `threading.RLock.acquire()`, starting with the name of the argument itself.

When invoked with the `block` argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the `timeout` argument are the same as in `Lock.acquire()`. Note that some of these behaviors of `timeout` differ from the implemented behaviors in `threading.RLock.acquire()`.

**release()**

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An `AssertionError` is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in `threading.RLock.release()`.
class multiprocessing.Semaphore([value])

A semaphore object: a close analog of threading.Semaphore.

A solitary difference from its close analog exists: its acquire method's first argument is named block, as is consistent with Lock.acquire().

**Note:** On Mac OS X, sem_timedwait is unsupported, so calling acquire() with a timeout will emulate that function's behavior using a sleeping loop.

**Note:** If the SIGINT signal generated by ctrl-c arrives while the main thread is blocked by a call to BoundedSemaphore.acquire(), Lock.acquire(), RLock.acquire(), Semaphore.acquire(), Condition.acquire() or Condition.wait() then the call will be immediately interrupted and KeyboardInterrupt will be raised.

This differs from the behaviour of threading where SIGINT will be ignored while the equivalent blocking calls are in progress.

**Note:** Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the multiprocessing.synchronize module will be disabled, and attempts to import it will result in an ImportError. See bpo-3770 for additional information.

**Shared ctypes Objects**

It is possible to create shared objects using shared memory which can
be inherited by child processes.

### multiprocessing.Value(typecode_or_type, *args, lock=True)

Return a ctypes object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the `value` attribute of a `Value`.

`typecode_or_type` determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the `array` module. `*args` is passed on to the constructor for the type.

If `lock` is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Operations like `+=` which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```python
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```python
with counter.get_lock():
    counter.value += 1
```

Note that `lock` is a keyword-only argument.
multiprocessing.\texttt{Array}(\texttt{typecode\_or\_type}, \texttt{size\_or\_initializer}, *, lock=True)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

\texttt{typecode\_or\_type} determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the \texttt{array} module. If \texttt{size\_or\_initializer} is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, \texttt{size\_or\_initializer} is a sequence which is used to initialize the array and whose length determines the length of the array.

If \texttt{lock} is \texttt{True} (the default) then a new lock object is created to synchronize access to the value. If \texttt{lock} is a \texttt{Lock} or \texttt{RLock} object then that will be used to synchronize access to the value. If \texttt{lock} is \texttt{False} then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that \texttt{lock} is a keyword only argument.

Note that an array of \texttt{ctypes.c\_char} has \texttt{value} and \texttt{raw} attributes which allow one to use it to store and retrieve strings.

The \texttt{multiprocessing.sharedctypes} module

The \texttt{multiprocessing.sharedctypes} module provides functions for allocating \texttt{ctypes} objects from shared memory which can be inherited by child processes.

\textbf{Note:} Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a
specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

`multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)`

Return a ctypes array allocated from shared memory.

`typecode_or_type` determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the `array` module. If `size_or_initializer` is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise `size_or_initializer` is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use `Array()` instead to make sure that access is automatically synchronized using a lock.

`multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)`

Return a ctypes object allocated from shared memory.

`typecode_or_type` determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the `array` module. `*args` is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use `Value()` instead to make sure that access is automatically synchronized using a lock.
Note that an array of `ctypes.c_char` has `value` and `raw` attributes which allow one to use it to store and retrieve strings – see documentation for `ctypes`.

`multiprocessing.sharedctypes.Array(typecode_or_type, size_or_initializer, *, lock=True)`

The same as `RawArray()` except that depending on the value of `lock` a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword-only argument.

`multiprocessing.sharedctypes.Value(typecode_or_type, *args, lock=True)`

The same as `RawValue()` except that depending on the value of `lock` a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword-only argument.

`multiprocessing.sharedctypes.copy(obj)`
Return a ctypes object allocated from shared memory which is a copy of the ctypes object `obj`.

`multiprocessing.sharedctypes.synchronized(obj[, lock])`

Return a process-safe wrapper object for a ctypes object which uses `lock` to synchronize access. If `lock` is `None` (the default) then a `multiprocessing.RLock` object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

*Changed in version 3.5:* Synchronized objects support the `context manager` protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of `ctypes.Structure`.)

<table>
<thead>
<tr>
<th>ctypes</th>
<th>sharedctypes using type</th>
<th>sharedctypes using typecode</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c_double(2.4)</code></td>
<td><code>RawValue(c_double, 2.4)</code></td>
<td><code>RawValue('d', 2.4)</code></td>
</tr>
<tr>
<td><code>MyStruct(4, 6)</code></td>
<td><code>RawValue(MyStruct, 4, 6)</code></td>
<td></td>
</tr>
<tr>
<td><code>(c_short * 7)()</code></td>
<td><code>RawArray(c_short, 7)</code></td>
<td><code>RawArray('h', 7)</code></td>
</tr>
<tr>
<td><code>(c_int * 3)(9, 2, 8)</code></td>
<td><code>RawArray(c_int, (9, 2, 8))</code></td>
<td><code>RawArray('i', (9, 2, 8))</code></td>
</tr>
</tbody>
</table>
Below is an example where a number of ctypes objects are modified by a child process:

```python
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('
x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875, -6.25), (-5.75, 2.0), (2

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([[a.x, a.y] for a in A])
```

The results printed are
Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

**multiprocessing.Manager()**

Returns a started `SyncManager` object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

```python
class multiprocessing.managers.BaseManager([address[, authkey]])
```

Create a BaseManager object.

Once created one should call `start()` or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

`address` is the address on which the manager process listens for
new connections. If `address` is `None` then an arbitrary one is chosen.

`authkey` is the authentication key which will be used to check the validity of incoming connections to the server process. If `authkey` is `None` then `current_process().authkey` is used. Otherwise `authkey` is used and it must be a byte string.

```
start([initializer, initargs])
```

Start a subprocess to start the manager. If `initializer` is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

```
get_server()
```

Returns a `Server` object which represents the actual server under the control of the Manager. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000),
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server additionally has an `address` attribute.

```
connect()
```

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000)
>>> m.connect()
```
**shutdown()**

Stop the process used by the manager. This is only available if **start()** has been used to start the server process.

This can be called multiple times.

**register(typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]])**

A classmethod which can be used for registering a type or callable with the manager class.

*typeid* is a “type identifier” which is used to identify a particular type of shared object. This must be a string.

*callable* is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the **connect()** method, or if the *create_method* argument is **False** then this can be left as **None**.

*proxytype* is a subclass of **BaseProxy** which is used to create proxies for shared objects with this *typeid*. If **None** then a proxy class is created automatically.

*exposed* is used to specify a sequence of method names which proxies for this typeid should be allowed to access using **BaseProxy._callmethod()**. (If *exposed* is **None** then *proxytype._exposed_* is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a **__call__()** method and whose name does not begin with ')._
method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If method_to_typeid is None then proxytype._method_to_typeid_ is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is None then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name typeid which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is True.

BaseManager instances also have one read-only property:

address
The address used by the manager.

Changed in version 3.3: Manager objects support the context management protocol – see Context Manager Types. __enter__() starts the server process (if it has not already started) and then returns the manager object. __exit__() calls shutdown().

In previous versions __enter__() did not start the manager's server process if it was not already started.

class multiprocessing.managers.SyncManager
A subclass of BaseManager which can be used for the synchronization of processes. Objects of this type are returned by multiprocessing.Manager().

Its methods create and return Proxy Objects for a number of
commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

**Barrier** *(parties[, action[, timeout]])*

Create a shared `threading.Barrier` object and return a proxy for it.

*New in version 3.3.*

**BoundedSemaphore** *([]*[value]*)*

Create a shared `threading.BoundedSemaphore` object and return a proxy for it.

**Condition** *([lock]*)

Create a shared `threading.Condition` object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a `threading.Lock` or `threading.RLock` object.

*Changed in version 3.3: The `wait_for()` method was added.*

**Event()**

Create a shared `threading.Event` object and return a proxy for it.

**Lock()**

Create a shared `threading.Lock` object and return a proxy for it.

**Namespace()**

Create a shared `Namespace` object and return a proxy for it.
Queue([maxsize])
  Create a shared queue.Queue object and return a proxy for it.

RLock()
  Create a shared threading.RLock object and return a proxy for it.

Semaphore([value])
  Create a shared threading.Semaphore object and return a proxy for it.

Array(typecode, sequence)
  Create an array and return a proxy for it.

Value(typecode, value)
  Create an object with a writable value attribute and return a proxy for it.

dict()
dict(mapping)
dict(sequence)
  Create a shared dict object and return a proxy for it.

list()
list(sequence)
  Create a shared list object and return a proxy for it.

Changed in version 3.6: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the SyncManager.
**class multiprocessing.managers.Namespace**

A type that can register with `SyncManager`.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `_` will be an attribute of the proxy and not an attribute of the referent:

```python
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3  # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

**Customized managers**

To create one's own manager, one creates a subclass of `BaseManager` and uses the `register()` classmethod to register new types or callables with the manager class. For example:

```python
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
```
pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))  # prints 7
        print(maths.mul(7, 8))  # prints 56

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```python
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```python
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000),)
>>> m.connect()
```
Another client can also use it:

```python
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000),
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```python
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
... QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```
Proxy Objects

A proxy is an object which refers to a shared object which lives (presumably) in a different process. The shared object is said to be the referent of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```python
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying `str()` to a proxy will return the representation of the referent, whereas applying `repr()` will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain Proxy Objects. This permits nesting of these managed lists, dicts, and other Proxy Objects:
```python
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)  # referent of a now contains referent of b
>>> print(a, b)
[ListProxy object, typeid 'list' at ...] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```python
>>> l_outer = manager.list([manager.dict() for i in range(2)])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) `list` or `dict` objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```python
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
```
This approach is perhaps less convenient than employing nested `Proxy Objects` for most use cases but also demonstrates a level of control over the synchronization.

**Note:** The proxy types in `multiprocessing` do nothing to support comparisons by value. So, for instance, we have:

```python
>>> manager.list([1, 2, 3]) == [1, 2, 3]
False
```

One should just use a copy of the referent instead when making comparisons.

```python
class multiprocessing.managers.BaseProxy
    Proxy objects are instances of subclasses of BaseProxy.

    _callmethod(methodname[, args[, kwds]])
        Call and return the result of a method of the proxy's referent.

        If proxy is a proxy whose referent is obj then the expression
        proxy._callmethod(methodname, args, kwds)
        will evaluate the expression
        getattr(obj, methodname)(*args, **kwds)
```
in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the `method_to_typeid` argument of `BaseManager.register()`.

If an exception is raised by the call, then is re-raised by `_callmethod()`. If some other exception is raised in the manager's process then this is converted into a RemoteError exception and is raised by `_callmethod()`.

Note in particular that an exception will be raised if `methodname` has not been exposed.

An example of the usage of `_callmethod()`:

```python
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),))
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))
Traceback (most recent call last):
  ...
IndexError: list index out of range
```

_**getvalue()**_

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

_**repr()**_

Return a representation of the proxy object.
`__str__()`

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

```python
class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]])
```

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

`processes` is the number of worker processes to use. If `processes` is `None` then the number returned by `os.cpu_count()` is used.

If `initializer` is not `None` then each worker process will call `initializer(*initargs)` when it starts.

`maxtasksperchild` is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default `maxtasksperchild` is `None`, which means worker processes will live
as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function multiprocessing.Pool() or the Pool() method of a context object. In both cases context is set appropriately.

Note that the methods of the pool object should only be called by the process which created the pool.

New in version 3.2: maxtasksperchild

New in version 3.4: context

Note: Worker processes within a Pool typically live for the complete duration of the Pool’s work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The maxtasksperchild argument to the Pool exposes this ability to the end user.

apply(func[, args[, kwds]])
Call func with arguments args and keyword arguments kwds. It blocks until the result is ready. Given this blocks, apply_async() is better suited for performing work in parallel. Additionally, func is only executed in one of the workers of the pool.

apply_async(func[, args[, kwds[, callback[, error_callback]]]])
A variant of the apply() method which returns a result object.
If `callback` is specified then it should be a callable which accepts a single argument. When the result becomes ready `callback` is applied to it, that is unless the call failed, in which case the `error_callback` is applied instead.

If `error_callback` is specified then it should be a callable which accepts a single argument. If the target function fails, then the `error_callback` is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

```python
map(func, iterable[, chunksize])
```
A parallel equivalent of the `map()` built-in function (it supports only one `iterable` argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting `chunksize` to a positive integer.

Note that it may cause high memory usage for very long iterables. Consider using `imap()` or `imap_unordered()` with explicit `chunksize` option for better efficiency.

```python
map_async(func, iterable[, chunksize[, callback[, error_callback]]]])
```
A variant of the `map()` method which returns a result object.

If `callback` is specified then it should be a callable which accepts a single argument. When the result becomes ready `callback` is applied to it, that is unless the call failed, in which
case the error_callback is applied instead.

If error_callback is specified then it should be a callable which accepts a single argument. If the target function fails, then the error_callback is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

```
imap(func, iterable[, chunksize])
```

A lazier version of `map()`.

The chunksize argument is the same as the one used by the `map()` method. For very long iterables using a large value for `chunksize` can make the job complete much faster than using the default value of 1.

Also if `chunksize` is 1 then the `next()` method of the iterator returned by the `imap()` method has an optional `timeout` parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within `timeout` seconds.

```
imap_unordered(func, iterable[, chunksize])
```

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

```
starmap(func, iterable[, chunksize])
```

Like `map()` except that the elements of the iterable are expected to be iterables that are unpacked as arguments.
Hence an iterable of \([(1, 2), (3, 4)]\) results in \([\text{func}(1, 2), \text{func}(3, 4)]\).

*New in version 3.3.*

**starmap_async**(*func*, *iterable*[ , *chunksize*[ , *callback*[ , *error_callback]*]]))

A combination of *starmap()* and *map_async()* that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

*New in version 3.3.*

**close()**

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

**terminate()**

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected *terminate()* will be called immediately.

**join()**

Wait for the worker processes to exit. One must call *close()* or *terminate()* before using *join()*.

*New in version 3.3:* Pool objects now support the context management protocol – see Context Manager Types. 

*\_\_enter\_\_()* returns the pool object, and *\_\_exit\_\_()* calls *terminate()*.

**class** *multiprocessing.pool.AsyncResult*
The class of the result returned by Pool.apply_async() and Pool.map_async().

**get([timeout])**
- Return the result when it arrives. If timeout is not None and the result does not arrive within timeout seconds then multiprocessing.TimeoutError is raised. If the remote call raised an exception then that exception will be reraised by get().

**wait([timeout])**
- Wait until the result is available or until timeout seconds pass.

**ready()**
- Return whether the call has completed.

**successful()**
- Return whether the call completed without raising an exception. Will raise AssertionError if the result is not ready.

The following example demonstrates the use of a pool:

```python
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:
        result = pool.apply_async(f, (10,))  # evaluate "f(10)" asynchronously in a single process
        print(result.get(timeout=1))  # prints "100" unless your computer is *very* slow

    print(pool.map(f, range(10)))  # prints "[0, 1, 4, ..., 81]"
```
it = pool.imap(f, range(10))
print(next(it))  # prints
print(next(it))  # prints
print(it.next(timeout=1))  # prints

result = pool.apply_async(time.sleep, (10,))
print(result.get(timeout=1))  # raises

Listeners and Clients

Usually message passing between processes is done using queues or by using Connection objects returned by Pipe().

However, the multiprocessing.connection module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for digest authentication using the hmac module, and for polling multiple connections at the same time.

multiprocessing.connection.deliver_challenge(connection, authkey)
    Send a randomly generated message to the other end of the connection and wait for a reply.

    If the reply matches the digest of the message using authkey as the key then a welcome message is sent to the other end of the connection. Otherwise AuthenticationError is raised.

multiprocessing.connection.answer_challenge(connection, authkey)
    Receive a message, calculate the digest of the message using authkey as the key, and then send the digest back.
If a welcome message is not received, then `AuthenticationError` is raised.

```python
multiprocessing.connection.Client(address[, family[, authkey]])
```

Attempt to set up a connection to the listener which is using address `address`, returning a `Connection`.

The type of the connection is determined by `family` argument, but this can generally be omitted since it can usually be inferred from the format of `address`. (See Address Formats)

If `authkey` is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if `authkey` is None. `AuthenticationError` is raised if authentication fails. See Authentication keys.

```python
class multiprocessing.connection.Listener([address[, family[, backlog[, authkey]]]])
```

A wrapper for a bound socket or Windows named pipe which is ‘listening’ for connections.

`address` is the address to be used by the bound socket or named pipe of the listener object.

**Note:** If an address of ‘0.0.0.0’ is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use ‘127.0.0.1’.

`family` is the type of socket (or named pipe) to use. This can be one of the strings `'AF_INET'` (for a TCP socket), `'AF_UNIX'` (for a
Unix domain socket) or 'AF_PIPE' (for a Windows named pipe). Of these only the first is guaranteed to be available. If family is None then the family is inferred from the format of address. If address is also None then a default is chosen. This default is the family which is assumed to be the fastest available. See Address Formats. Note that if family is 'AF_UNIX' and address is None then the socket will be created in a private temporary directory created using tempfile.mkstemp().

If the listener object uses a socket then backlog (1 by default) is passed to the listen() method of the socket once it has been bound.

If authkey is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if authkey is None. AuthenticationError is raised if authentication fails. See Authentication keys.

**accept()**

Accept a connection on the bound socket or named pipe of the listener object and return a Connection object. If authentication is attempted and fails, then AuthenticationError is raised.

**close()**

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

**address**
The address which is being used by the Listener object.

**lastaccepted**

The address from which the last accepted connection came. If this is unavailable then it is None.

*New in version 3.3:* Listener objects now support the context management protocol – see [Context Manager Types](#). `__enter__()` returns the listener object, and `__exit__()` calls `close()`.

`multiprocessing.connection.wait(object_list, timeout=None)`

Wait till an object in `object_list` is ready. Returns the list of those objects in `object_list` which are ready. If `timeout` is a float then the call blocks for at most that many seconds. If `timeout` is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both Unix and Windows, an object can appear in `object_list` if it is

- a readable `Connection` object;
- a connected and readable `socket.socket` object; or
- the `sentinel` attribute of a `Process` object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

*Unix:* `wait(object_list, timeout)` almost equivalent to `select.select(object_list, [], [], timeout)`. The difference is that, if `select.select()` is interrupted by a signal, it can raise `OSError` with an error number of EINTR, whereas `wait()` will not.
**Windows**: An item in `object_list` must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

*New in version 3.3.*

**Examples**

The following server code creates a listener which uses `'secret password'` as an authentication key. It then waits for a connection and sends some data to the client:

```python
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)  # family is deduced

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])
	n conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv()) # => [2.25, None, 'junk', float]
    print(conn.recv_bytes()) # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print(conn.recv_bytes_into(arr)) # => 8
print(arr) # => array('i', [42, 1729, 0, 0, 0])

The following code uses `wait()` to wait for messages from multiple processes at once:

import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now
        # p is the only process which owns a handle
# ensures that when p closes its handle for
# wait() will promptly report the readable end
w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)

Address Formats

- An 'AF_INET' address is a tuple of the form (hostname, port) where hostname is a string and port is an integer.
- An 'AF_UNIX' address is a string representing a filename on the filesystem.
- An 'AF_PIPE' address is a string of the form \pipe{PipeName}. To use Client() to connect to a named pipe on a remote computer called ServerName one should use an address of the form \ServerName\pipe{PipeName} instead.

Note that any string beginning with two backslashes is assumed by default to be an 'AF_PIPE' address rather than an 'AF_UNIX' address.

Authentication keys

When one uses Connection.recv, the data received is automatically unpickled. Unfortunately unpickling data from an
untrusted source is a security risk. Therefore **Listener** and **Client()** use the **hmac** module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of **current_process().authkey** is used (see **Process**). This value will be automatically inherited by any **Process** object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using **os.urandom()**.

**Logging**

Some support for logging is available. Note, however, that the **logging** package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

**multiprocessing.get_logger()**

Returns the logger used by **multiprocessing**. If necessary, a new one will be created.

When first created the logger has level **logging.NOTSET** and no
default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

**multiprocessing.log_to_stderr()**

This function performs a call to `get_logger()` but in addition to returning the logger created by `get_logger`, it adds a handler which sends output to `sys.stderr` using format `[%(levelname)s/%(processName)s] %(message)s`.

Below is an example session with logging turned on:

```python
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-....] child process calling self.run()
[INFO/SyncManager-....] created temp directory /.../p
[INFO/SyncManager-....] manager serving at '/.../listener-

For a full table of logging levels, see the `logging` module.

The **multiprocessing.dummy** module

`multiprocessing.dummy` replicates the API of `multiprocessing`
but is no more than a wrapper around the `threading` module.
Programming guidelines

There are certain guidelines and idioms which should be adhered to when using multiprocessing.

All start methods

The following applies to all start methods.

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the same proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it
becomes a zombie. There should never be very many because each time a new process starts (or \texttt{active_children()} is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's \texttt{Process.is_alive} will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

**Better to inherit than pickle/unpickle**

When using the \texttt{spawn} or \texttt{forkserver} start methods many types from \texttt{multiprocessing} need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

**Avoid terminating processes**

Using the \texttt{Process.terminate} method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using \texttt{Process.terminate} on processes which never use any shared resources.

**Joining processes that use queues**

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the \texttt{Queue.cancel_join_thread} method of the queue to
avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```python
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()  # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On Unix using the `fork` start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows
and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```python
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```python
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

```python
import multiprocessing

multiprocessing
```

```python
os.close(sys.stdin.fileno())
```
in the multiprocessing.Process._bootstrap() method — this resulted in issues with processes-in-processes. This has been changed to:

```python
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY),)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a “file-like object” with output buffering. This danger is that if multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```python
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155, bpo-5313 and bpo-5331

The `spawn` and `forkserver` start methods

There are a few extra restriction which don’t apply to the `fork` start
method.

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. Also, if you subclass `Process` then make sure that instances will be picklable when the `Process.start` method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that `Process.start` was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).

For example, using the `spawn` or `forkserver` start method running the following module would fail with a `RuntimeError`:

```python
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```
Instead one should protect the “entry point” of the program by using `if __name__ == '__main__':` as follows:

```python
from multiprocessing import Process, freeze_support

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module’s `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.
Examples

Demonstration of how to create and use customized managers and proxies:

```python
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def __h__(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator
```
class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
assert not hasattr(f2, 'f')
assert sorted(f2._exposed_) == sorted(['g', '_h'])

print('-' * 20)

it = manager.baz()
for i in it:
    print('<%d>' % i, end=' ')  
print()

print('-' * 20)

op = manager.operator()
print('op.add(23, 45) =', op.add(23, 45))
print('op.pow(2, 94) =', op.pow(2, 94))
print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()

Using **Pool**:

```python
import multiprocessing
import time
import random
import sys

# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
```
return '%%s says that %%s%%s = %%s' % (multiprocessing.current_process().name, func.__name__, args, result)

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

# # Test code # #

def test():
    PROCESSES = 4
    print('Creating pool with %d processes

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #
TASKS = [(mul, (i, 7)) for i in range(10)] +
        [(plus, (i, 8)) for i in range(10)]

results = [pool.apply_async(calculate, t) for t in TASKS]
imap_it = pool.imap(calculatestar, TASKS)
imap_unordered_it = pool.imap_unordered(calculatestar)

print('Ordered results using pool.apply_async():')
for r in results:
    print('	', r.get())
print()

print('Ordered results using pool.imap():')
for x in imap_it:
    print('	', x)
print()

print('Unordered results using pool.imap_unordered():')
for x in imap_unordered_it:
    print('	', x)
print()

print('Ordered results using pool.map() ---')
for x in pool.map(calculatestar, TASKS):
    print('	', x)
print()

#
# Test error handling
#

print('Testing error handling:')

try:
    print(pool.apply(f, (5,)))
except ZeroDivisionError:
print('\tGot ZeroDivisionError as expected from pool.apply()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(pool.map(f, list(range(10))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.map()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(list(pool.imap(f, list(range(10)))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from list(pool.imap())')
else:
    raise AssertionError('expected ZeroDivisionError')

it = pool.imap(f, list(range(10)))
for i in range(10):
    try:
        x = next(it)
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

assert i == 9
print('\tGot ZeroDivisionError as expected for')
print()

#  # Testing timeouts
print('Testing ApplyResult.get() with timeout:')
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('
	%s' % res.get()
    except multiprocessing.TimeoutError:
        break
print()
print()

print('Testing IMapIterator.next() with timeout:')
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('
	%s' % it.next()
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.
')
    print()
print()

if __name__ == '__main__':
multiprocessing.freeze_support()
test()

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

import time
import random

from multiprocessing import Process, Queue, current_process

# Function run by worker processes
#
def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

# Function used to calculate result
#
def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % 
    (current_process().name, func.__name__, args)

# Functions referenced by tasks
#
def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

# 
def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results: '
    for i in range(len(TASKS1)):
        print('	', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('	', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')

if __name__ == '__main__':
The concurrent package

Currently, there is only one module in this package:

- `concurrent.futures` – Launching parallel tasks
The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using `ThreadPoolExecutor`, or separate processes, using `ProcessPoolExecutor`. Both implement the same interface, which is defined by the abstract `Executor` class.
Executor Objects

`class concurrent.futures.Executor`

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

`submit(fn, *args, **kwargs)`

Schedules the callable, `fn`, to be executed as `fn(*args, **kwargs)` and returns a `Future` object representing the execution of the callable.

```python
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

`map(func, *iterables, timeout=None, chunksize=1)`

Similar to `map(func, *iterables)` except:

- the `iterables` are collected immediately rather than lazily;
- `func` is executed asynchronously and several calls to `func` may be made concurrently.

The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn’t available after `timeout` seconds from the original call to `Executor.map()`. `timeout` can be an int or a float. If `timeout` is not specified or `None`, there is no limit to the wait time.

If a `func` call raises an exception, then that exception will be
raised when its value is retrieved from the iterator.

When using `ProcessPoolExecutor`, this method chops `iterables` into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting `chunksize` to a positive integer. For very long iterables, using a large value for `chunksize` can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor`, `chunksize` has no effect.

*Changed in version 3.5:* Added the `chunksize` argument.

**shutdown** *(wait=True)*

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If `wait` is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If `wait` is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of `wait`, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were
called with `wait` set to `True`):

```python
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```
ThreadExecutor

ThreadExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a Future waits on the results of another Future. For example:

```python
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because...
    return 5
def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because...
    return 6

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```python
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```
class concurrent.futures.ThreadPoolExecutor(max_workers=None, thread_name_prefix='', initializer=None, initargs=())

An Executor subclass that uses a pool of at most max_workers threads to execute calls asynchronously.

initializer is an optional callable that is called at the start of each worker thread; initargs is a tuple of arguments passed to the initializer. Should initializer raise an exception, all currently pending jobs will raise a BrokenThreadPool, as well as any attempt to submit more jobs to the pool.

Changed in version 3.5: If max_workers is None or not given, it will default to the number of processors on the machine, multiplied by 5, assuming that ThreadPoolExecutor is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for ProcessPoolExecutor.

New in version 3.6: The thread_name_prefix argument was added to allow users to control the threading.Thread names for worker threads created by the pool for easier debugging.

Changed in version 3.7: Added the initializer and initargs arguments.

ThreadPoolExecutor Example

```python
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
]```
Retrieve a single page and report the URL and contents

def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout):
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=):
    # Start the load operations and mark each future
    future_to_url = {executor.submit(load_url, url, url):
        for future in concurrent.futures.as_completed(future_to_url):
            url = future_to_url[future]
            try:
                data = future.result()
            except Exception as exc:
                print('%r generated an exception: %s' % (url, exc))
            else:
                print('%r page is %d bytes' % (url, len(data))
ProcessPoolExecutor

The `ProcessPoolExecutor` class is an `Executor` subclass that uses a pool of processes to execute calls asynchronously. `ProcessPoolExecutor` uses the `multiprocessing` module, which allows it to side-step the `Global Interpreter Lock` but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that `ProcessPoolExecutor` will not work in the interactive interpreter.

Calling `Executor` or `Future` methods from a callable submitted to a `ProcessPoolExecutor` will result in deadlock.

```python
class concurrent.futures.ProcessPoolExecutor(max_workers=None, mp_context=None, initializer=None, initargs=())
```

An `Executor` subclass that executes calls asynchronously using a pool of at most `max_workers` processes. If `max_workers` is `None` or not given, it will default to the number of processors on the machine. If `max_workers` is lower or equal to `0`, then a `ValueError` will be raised. `mp_context` can be a multiprocessing context or `None`. It will be used to launch the workers. If `mp_context` is `None` or not given, the default multiprocessing context is used.

`initializer` is an optional callable that is called at the start of each worker process; `initargs` is a tuple of arguments passed to the initializer. Should `initializer` raise an exception, all currently pending jobs will raise a `BrokenProcessPool`, as well any attempt to
submit more jobs to the pool.

*Changed in version 3.3:* When one of the worker processes terminates abruptly, a BrokenProcessPool error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

*Changed in version 3.7:* The `mp_context` argument was added to allow users to control the `start_method` for worker processes created by the pool.

Added the `initializer` and `initargs` arguments.

**ProcessPoolExecutor Example**

```python
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False

    return True
```
```python
def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(print, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```
Future Objects

The `Future` class encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()`.

```python
class concurrent.futures.Future
    Encapsulates the asynchronous execution of a callable. Future instances are created by `Executor.submit()` and should not be created directly except for testing.
```

cancel()
    Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()
    Return `True` if the call was successfully cancelled.

running()
    Return `True` if the call is currently being executed and cannot be cancelled.

done()
    Return `True` if the call was successfully cancelled or finished running.

result(timeout=None)
    Return the value returned by the call. If the call hasn’t yet completed then this method will wait up to `timeout` seconds. If the call hasn’t completed in `timeout` seconds,
then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised, this method will raise the same exception.

`exception(timeout=None)`

Return the exception raised by the call. If the call hasn’t yet completed then this method will wait up to *timeout* seconds. If the call hasn’t completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

`add_done_callback(fn)`

Attaches the callable `fn` to the future. `fn` will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.
If the future has already completed or been cancelled, \( fn \) will be called immediately.

The following \texttt{Future} methods are meant for use in unit tests and \texttt{Executor} implementations.

\textbf{set\_running\_or\_notify\_cancel()}

This method should only be called by \texttt{Executor} implementations before executing the work associated with the \texttt{Future} and by unit tests.

If the method returns \texttt{False} then the \texttt{Future} was cancelled, i.e. \texttt{Future.cancel()} was called and returned \texttt{True}. Any threads waiting on the \texttt{Future} completing (i.e. through \texttt{as\_completed()} or \texttt{wait()}) will be woken up.

If the method returns \texttt{True} then the \texttt{Future} was not cancelled and has been put in the running state, i.e. calls to \texttt{Future.running()} will return \texttt{True}.

This method can only be called once and cannot be called after \texttt{Future.set\_result()} or \texttt{Future.set\_exception()} have been called.

\textbf{set\_result}(\texttt{result})

Sets the result of the work associated with the \texttt{Future} to \texttt{result}.

This method should only be used by \texttt{Executor} implementations and unit tests.

\textbf{set\_exception}(\texttt{exception})
Sets the result of the work associated with the Future to the Exception exception.

This method should only be used by Executor implementations and unit tests.
Module Functions

concurrent.futures.wait(fs, timeout=None, return_when=ALL_COMPLETED)

Wait for the Future instances (possibly created by different Executor instances) given by fs to complete. Returns a named 2-tuple of sets. The first set, named done, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named not_done, contains uncompleted futures.

timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST_COMPLETED</td>
<td>The function will return when any future finishes or is cancelled.</td>
</tr>
<tr>
<td>FIRST_EXCEPTION</td>
<td>The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.</td>
</tr>
<tr>
<td>ALL_COMPLETED</td>
<td>The function will return when all futures finish or are cancelled.</td>
</tr>
</tbody>
</table>

concurrent.futures.as_completed(fs, timeout=None)

Returns an iterator over the Future instances (possibly created by different Executor instances) given by fs that yields futures as
they complete (finished or were cancelled). Any futures given by `fs` that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn’t available after `timeout` seconds from the original call to `as_completed()`. `timeout` can be an int or float. If `timeout` is not specified or `None`, there is no limit to the wait time.

See also:

**PEP 3148 – futures - execute computations asynchronously**

The proposal which described this feature for inclusion in the Python standard library.
Exception classes

`exception concurrent.futures.CancelledError`
Raised when a future is cancelled.

`exception concurrent.futures.TimeoutError`
Raised when a future operation exceeds the given timeout.

`exception concurrent.futures.BrokenExecutor`
Derived from `RuntimeError`, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

*New in version 3.7.*

`exception concurrent.futures.thread.BrokenThreadPool`
Derived from `BrokenExecutor`, this exception class is raised when one of the workers of a `ThreadPoolExecutor` has failed initializing.

*New in version 3.7.*

`exception concurrent.futures.process.BrokenProcessPool`
Derived from `BrokenExecutor` (formerly `RuntimeError`), this exception class is raised when one of the workers of a `ProcessPoolExecutor` has terminated in a non-clean fashion (for example, if it was killed from the outside).

*New in version 3.3.*
The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```python
os.system
os.spawn`
```

Information about how the `subprocess` module can be used to replace these modules and functions can be found in the following sections.

**See also:** [PEP 324](https://www.python.org/dev/peps/pep-0324) – PEP proposing the subprocess module
Using the `subprocess` Module

The recommended approach to invoking subprocesses is to use the `run()` function for all use cases it can handle. For more advanced use cases, the underlying `Popen` interface can be used directly.

The `run()` function was added in Python 3.5; if you need to retain compatibility with older versions, see the Older high-level API section.

```python
subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None)
```

Run the command described by `args`. Wait for command to complete, then return a `CompletedProcess` instance.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - most of the arguments to this function are passed through to that interface. (`timeout`, `input`, `check`, and `capture_output` are not.)

If `capture_output` is true, stdout and stderr will be captured. When used, the internal `Popen` object is automatically created with `stdout=PIPE` and `stderr=PIPE`. The `stdout` and `stderr` arguments may not be used as well.

The `timeout` argument is passed to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child
process has terminated.

The *input* argument is passed to *Popen.communicate()* and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal *Popen* object is automatically created with stdin=PIPE, and the stdin argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a *CalledProcessError* exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the *io.TextIOWrapper* default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to *Popen*.

Examples:

```python
g>> subprocess.run(['ls', '-l'])  # doesn't capture
CompletedProcess(args=['ls', '-l'], returncode=0)

>> subprocess.run('exit 1', shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' r
```python
>>> subprocess.run(['ls', '-l', '/dev/null'], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'],
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null
', stderr=b'')
```

New in version 3.5.

Changed in version 3.6: Added encoding and errors parameters

Changed in version 3.7: Added the text parameter, as a more understandable alias of universal_newlines. Added the capture_output parameter.

class subprocess.CompletedProcess
The return value from `run()`, representing a process that has finished.

args
The arguments used to launch the process. This may be a list or a string.

returncode
Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value -N indicates that the child was terminated by signal N (POSIX only).

stdout
Captured stdout from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or text=True. None if stdout was not captured.
If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and `stderr` will be None.

**stderr**
Captured stderr from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or `text=True`. None if stderr was not captured.

**check_returncode()**
If `returncode` is non-zero, raise a `CalledProcessError`.

*New in version 3.5.*

**subprocess.DEVNULL**
Special value that can be used as the `stdin`, `stdout` or `stderr` argument to `Popen` and indicates that the special file `os.devnull` will be used.

*New in version 3.3.*

**subprocess.PIPE**
Special value that can be used as the `stdin`, `stdout` or `stderr` argument to `Popen` and indicates that a pipe to the standard stream should be opened. Most useful with `Popen.communicate()`.

**subprocess.STDOUT**
Special value that can be used as the `stderr` argument to `Popen` and indicates that standard error should go into the same handle as standard output.

*exception* subprocess.**SubprocessError**
Base class for all other exceptions from this module.

*New in version 3.3.*

**exception** `subprocess.TimeoutExpired`  
Subclass of `SubprocessError`, raised when a timeout expires while waiting for a child process.

**cmd**  
Command that was used to spawn the child process.

**timeout**  
Timeout in seconds.

**output**  
Output of the child process if it was captured by `run()` or `check_output()`. Otherwise, `None`.

**stdout**  
Alias for `output`, for symmetry with `stderr`.

**stderr**  
Stderr output of the child process if it was captured by `run()`. Otherwise, `None`.

*New in version 3.3.*

*Changed in version 3.5: stdout and stderr attributes added*

**exception** `subprocess.CalledProcessError`  
Subclass of `SubprocessError`, raised when a process run by `check_call()` or `check_output()` returns a non-zero exit status.
**returncode**

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

**cmd**

Command that was used to spawn the child process.

**output**

Output of the child process if it was captured by `run()` or `check_output()`. Otherwise, None.

**stdout**

Alias for output, for symmetry with `stderr`.

**stderr**

Stderr output of the child process if it was captured by `run()`. Otherwise, None.

*Changed in version 3.5: stdout and stderr attributes added*

### Frequently Used Arguments

To support a wide variety of use cases, the `Popen` constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

- **args** is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit
spaces in file names). If passing a single string, either shell must be `True` (see below) or else the string must simply name the program to be executed without specifying any arguments.

`stdin`, `stdout` and `stderr` specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, `stderr` can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for `stdout`.

If `encoding` or `errors` are specified, or `text` (also known as `universal_newlines`) is true, the file objects `stdin`, `stdout` and `stderr` will be opened in text mode using the `encoding` and `errors` specified in the call or the defaults for `io.TextIOWrapper`.

For `stdin`, line ending characters `\n` in the input will be converted to the default line separator `os.linesep`. For `stdout` and `stderr`, all line endings in the output will be converted to `\n`. For more information see the documentation of the `io.TextIOWrapper` class when the `newline` argument to its constructor is `None`.

If text mode is not used, `stdin`, `stdout` and `stderr` will be opened as binary streams. No encoding or line ending conversion is performed.
New in version 3.6: Added `encoding` and `errors` parameters.

New in version 3.7: Added the `text` parameter as an alias for `universal_newlines`.

**Note:** The newlines attribute of the file objects `Popen.stdin`, `Popen.stdout` and `Popen.stderr` are not updated by the `Popen.communicate()` method.

If `shell` is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user’s home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`).

*Changed in version 3.3:* When `universal_newlines` is `True`, the class uses the encoding `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`.

**Note:** Read the Security Considerations section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the `Popen` constructor documentation.
Popen Constructor

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

```python
class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=None, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=(), *, encoding=None, errors=None, text=None)
```

Execute a child program in a new process. On POSIX, the class uses `os.execvp()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.

`args` should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in `args` if `args` is a sequence. If `args` is a string, the interpretation is platform-dependent and described below. See the `shell` and `executable` arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass `args` as a sequence.

On POSIX, if `args` is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

**Note:** `shlex.split()` can be useful when determining the correct tokenization for `args`, especially in complex cases:
Note in particular that options (such as `-input`) and arguments (such as `eggs.txt`) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the `echo` command shown above) are single list elements.

On Windows, if `args` is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying `CreateProcess()` operates on strings.

The `shell` argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If `shell` is `True`, it is recommended to pass `args` as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If `args` is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If `args` is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, `Popen` does the equivalent of:
Popen(["/bin/sh", '-c', args[0], args[1], ...])

On Windows with `shell=True`, the COMSPEC environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. `dir` or `copy`). You do not need `shell=True` to run a batch file or console-based executable.

**Note:** Read the [Security Considerations](#) section before using `shell=True`.

`bufsize` will be supplied as the corresponding argument to the `open()` function when creating the stdin/stdout/stderr pipe file objects:

- 0 means unbuffered (read and write are one system call and can return short)
- 1 means line buffered (only usable if `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of `io.DEFAULT_BUFFER_SIZE` will be used.

*Changed in version 3.3.1:* bufsize now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to 0 which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The `executable` argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`,
executable replaces the program to execute specified by args. However, the original args is still passed to the program. Most programs treat the program specified by args as the command name, which can then be different from the program actually executed. On POSIX, the args name becomes the display name for the executable in utilities such as ps. If shell=True, on POSIX the executable argument specifies a replacement shell for the default /bin/sh.

stdin, stdout and stderr specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are PIPE, DEVNULL, an existing file descriptor (a positive integer), an existing file object, and None. PIPE indicates that a new pipe to the child should be created. DEVNULL indicates that the special file os.devnull will be used. With the default settings of None, no redirection will occur; the child’s file handles will be inherited from the parent. Additionally, stderr can be STDOUT, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If preexec_fn is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

**Warning:** The preexec_fn parameter is not safe to use in the presence of threads in your application. The child process could deadlock before exec is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

**Note:** If you need to modify the environment for the child use the env parameter rather than doing it in a preexec_fn. The start_new_session parameter can take the place of a previously common use of preexec_fn to call os.setsid() in the child.
If `close_fds` is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. Otherwise when `close_fds` is false, file descriptors obey their inheritable flag as described in Inheritance of File Descriptors.

On Windows, if `close_fds` is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of `STARTUPINFO.lpAttributeList`, or by standard handle redirection.

*Changed in version 3.2:* The default for `close_fds` was changed from `False` to what is described above.

*Changed in version 3.7:* On Windows the default for `close_fds` was changed from `False` to `True` when redirecting the standard handles. It’s now possible to set `close_fds` to `True` when redirecting the standard handles.

`pass_fds` is an optional sequence of file descriptors to keep open between the parent and child. Providing any `pass_fds` forces `close_fds` to be `True`. (POSIX only)

*New in version 3.2:* The `pass_fds` parameter was added.

If `cwd` is not `None`, the function changes the working directory to `cwd` before executing the child. `cwd` can be a `str` and path-like object. In particular, the function looks for `executable` (or for the first item in `args`) relative to `cwd` if the executable path is a relative path.

*Changed in version 3.6:* `cwd` parameter accepts a path-like object.

If `restore_signals` is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before
the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

*Changed in version 3.2:* `restore_signals` was added.

If `start_new_session` is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess. (POSIX only)

*Changed in version 3.2:* `start_new_session` was added.

If `env` is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process’ environment.

**Note:** If specified, `env` must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified `env` must include a valid `SystemRoot`.

If `encoding` or `errors` are specified, or `text` is true, the file objects `stdin`, `stdout` and `stderr` are opened in text mode with the specified encoding and `errors`, as described above in *Frequently Used Arguments*. The `universal_newlines` argument is equivalent to `text` and is provided for backwards compatibility. By default, file objects are opened in binary mode.

*New in version 3.6:* `encoding` and `errors` were added.

*New in version 3.7:* `text` was added as a more readable alias for `universal_newlines`.

If given, `startupinfo` will be a `STARTUPINFO` object, which is
passed to the underlying `CreateProcess` function. `creationflags`, if given, can be one or more of the following flags:

- `CREATE_NEW_CONSOLE`
- `CREATE_NEW_PROCESS_GROUP`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `IDLE_PRIORITY_CLASS`
- `NORMAL_PRIORITY_CLASS`
- `REALTIME_PRIORITY_CLASS`
- `CREATE_NO_WINDOW`
- `DETACHED_PROCESS`
- `CREATE_DEFAULT_ERROR_MODE`
- `CREATE_BREAKAWAY_FROM_JOB`

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```python
with Popen(['ifconfig'], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

*Changed in version 3.2:* Added context manager support.

*Changed in version 3.6:* Popen destructor now emits a `ResourceWarning` warning if the child process is still running.

**Exceptions**

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.
The most common exception raised is \texttt{OSError}. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for \texttt{OSError} exceptions.

A \texttt{ValueError} will be raised if \texttt{Popen} is called with invalid arguments.

\texttt{check\_call()} and \texttt{check\_output()} will raise \texttt{CalledProcessError} if the called process returns a non-zero return code.

All of the functions and methods that accept a \textit{timeout} parameter, such as \texttt{call()} and \texttt{Popen.communicate()} will raise \texttt{TimeoutExpired} if the timeout expires before the process exits.

Exceptions defined in this module all inherit from \texttt{SubprocessError}.

\textit{New in version 3.3:} The \texttt{SubprocessError} base class was added.
Security Considerations

Unlike some other popen functions, this implementation will never implicitly call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application’s responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities.

When using `shell=True`, the `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.
Popen Objects

Instances of the `Popen` class have the following methods:

`Popen.poll()`
Check if child process has terminated. Set and return `returncode` attribute. Otherwise, returns `None`.

`Popen.wait(timeout=None)`
Wait for child process to terminate. Set and return `returncode` attribute.

If the process does not terminate after `timeout` seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.

**Note:** This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `Popen.communicate()` when using pipes to avoid that.

**Note:** The function is implemented using a busy loop (non-blocking call and short sleeps). Use the `asyncio` module for an asynchronous wait: see `asyncio.create_subprocess_exec`.

`Changed in version 3.3: timeout was added.`

`Popen.communicate(input=None, timeout=None)`
Interact with process: Send data to stdin. Read data from stdout
and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or *None*, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process’s stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than *None* in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a `TimeoutExpired` exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```python
proc = subprocess.Popen(...)  
try:  
    outs, errs = proc.communicate(timeout=15)  
except TimeoutExpired:  
    proc.kill()  
    outs, errs = proc.communicate()
```

**Note:** The data read is buffered in memory, so do not use this method if the data size is large or unlimited.
Changed in version 3.3: timeout was added.

Popen. `send_signal(signal)`

Sends the signal `signal` to the child.

**Note:** On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a `creationflags` parameter which includes `CREATE_NEW_PROCESS_GROUP`.

Popen. `terminate()`

Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function TerminateProcess() is called to stop the child.

Popen. `kill()`

Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows `kill()` is an alias for `terminate()`.

The following attributes are also available:

Popen. `args`

The `args` argument as it was passed to `Popen` – a sequence of program arguments or else a single string.

*New in version 3.3.*

Popen. `stdin`

If the `stdin` argument was `PIPE`, this attribute is a writeable stream object as returned by `open()`. If the `encoding` or `errors` arguments were specified or the `universal_newlines` argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the `stdin`
argument was not PIPE, this attribute is None.

Popen. **stdout**

If the *stdout* argument was PIPE, this attribute is a readable stream object as returned by open(). Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was True, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not PIPE, this attribute is None.

Popen. **stderr**

If the *stderr* argument was PIPE, this attribute is a readable stream object as returned by open(). Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was True, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not PIPE, this attribute is None.

**Warning:** Use communicate() rather than .stdin.write, .stdout.read or .stderr.read to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen. **pid**

The process ID of the child process.

Note that if you set the *shell* argument to True, this is the process ID of the spawned shell.

Popen. **returncode**

The child return code, set by poll() and wait() (and indirectly by communicate()). A None value indicates that the process hasn't terminated yet.
A negative value -N indicates that the child was terminated by signal N (POSIX only).
Windows Popen Helpers

The **STARTUPINFO** class and following constants are only available on Windows.

```python
class subprocess.STARTUPINFO(*, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None)
```

Partial support of the Windows **STARTUPINFO** structure is used for **Popen** creation. The following attributes can be set by passing them as keyword-only arguments.

*Changed in version 3.7:* Keyword-only argument support was added.

**dwFlags**

A bit field that determines whether certain **STARTUPINFO** attributes are used when the process creates a window.

```python
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES |
```

**hStdInput**

If **dwFlags** specifies **STARTF_USESTDHANDLES**, this attribute is the standard input handle for the process. If **STARTF_USESTDHANDLES** is not specified, the default for standard input is the keyboard buffer.

**hStdOutput**

If **dwFlags** specifies **STARTF_USESTDHANDLES**, this attribute is the standard output handle for the process. Otherwise, this
attribute is ignored and the default for standard output is the console window’s buffer.

**hStdError**

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window’s buffer.

**wShowWindow**

If `dwFlags` specifies `STARTF_USESHOWWINDOW`, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the `ShowWindow` function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

`SW_HIDE` is provided for this attribute. It is used when `Popen` is called with `shell=True`.

**lpAttributeList**

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see `UpdateProcThreadAttribute`.

Supported attributes:

**handle_list**

Sequence of handles that will be inherited. `close_fds` must be true if non-empty.

The handles must be temporarily made inheritable by `os.set_handle_inheritable()` when passed to the `Popen` constructor, else `OSError` will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).
Warning: In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as `os.system()`. This also applies to standard handle redirection, which temporarily creates inheritable handles.

New in version 3.7.

Windows Constants

The `subprocess` module exposes the following constants.

`subprocess.STD_INPUT_HANDLE`  
The standard input device. Initially, this is the console input buffer, `CONIN$`.

`subprocess.STD_OUTPUT_HANDLE`  
The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.STD_ERROR_HANDLE`  
The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.SW_HIDE`  
Hides the window. Another window will be activated.

`subprocess.STARTF_USESTDHANDLES`  
Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.
subprocess.**STARTF_USESHOWWINDOW**
Specifications that the **STARTUPINFO.wShowWindow** attribute contains additional information.

subprocess.**CREATE_NEW_CONSOLE**
The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.**CREATE_NEW_PROCESS_GROUP**
A **Popen creationflags** parameter to specify that a new process group will be created. This flag is necessary for using **os.kill()** on the subprocess.

This flag is ignored if **CREATE_NEW_CONSOLE** is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**
A **Popen creationflags** parameter to specify that a new process will have an above average priority.

*New in version 3.7.*

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**
A **Popen creationflags** parameter to specify that a new process will have a below average priority.

*New in version 3.7.*

subprocess.**HIGH_PRIORITY_CLASS**
A **Popen creationflags** parameter to specify that a new process will have a high priority.

*New in version 3.7.*

subprocess.**IDLE_PRIORITY_CLASS**
A `Popen` creationflags parameter to specify that a new process will have an idle (lowest) priority.

*New in version 3.7.*

subprocess.\texttt{NORMAL\_PRIORITY\_CLASS}

A `Popen` creationflags parameter to specify that a new process will have an normal priority. (default)

*New in version 3.7.*

subprocess.\texttt{REALTIME\_PRIORITY\_CLASS}

A `Popen` creationflags parameter to specify that a new process will have realtime priority. You should almost never use \texttt{REALTIME\_PRIORITY\_CLASS}, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that “talk” directly to hardware or that perform brief tasks that should have limited interruptions.

*New in version 3.7.*

subprocess.\texttt{CREATE\_NO\_WINDOW}

A `Popen` creationflags parameter to specify that a new process will not create a window.

*New in version 3.7.*

subprocess.\texttt{DETACHED\_PROCESS}

A `Popen` creationflags parameter to specify that a new process will not inherit its parent’s console. This value cannot be used with \texttt{CREATE\_NEW\_CONSOLE}.

*New in version 3.7.*
subprocess. `CREATE_DEFAULT_ERROR_MODE`

A `Popen` `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

*New in version 3.7.*

subprocess. `CREATE_BREAKAWAY_FROM_JOB`

A `Popen` `creationflags` parameter to specify that a new process is not associated with the job.

*New in version 3.7.*
Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use `run()` in many cases, but lots of existing code calls these functions.

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)
```

Run the command described by `args`. Wait for command to complete, then return the `returncode` attribute.

This is equivalent to:

```
run(...).returncode
```

(except that the `input` and `check` parameters are not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the `Popen` constructor - this function passes all supplied arguments other than `timeout` directly through to that interface.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

*Changed in version 3.3:* `timeout` was added.

```
subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None)
```

Run command with arguments. Wait for command to complete. If
the return code was zero then return, otherwise raise
`CalledProcessError`. The `CalledProcessError` object will
have the return code in the `returncode` attribute.

This is equivalent to:

```python
text
run(..., check=True)
```

(except that the `input` parameter is not supported)

The arguments shown above are merely the most common ones.
The full function signature is largely the same as that of the `Popen`
constructor - this function passes all supplied arguments other than `timeout` directly through to that interface.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this
function. The child process will block if it generates enough
output to a pipe to fill up the OS pipe buffer as the pipes are not
being read from.

*Changed in version 3.3: timeout was added.*

```python
text
subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, universal_newlines=None, timeout=None, text=None)
```

Run command with arguments and return its output.

If the return code was non-zero it raises a
`CalledProcessError`. The `CalledProcessError` object will
have the return code in the `returncode` attribute and any output
in the `output` attribute.

This is equivalent to:
run(..., check=True, stdout=PIPE).stdout

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of \texttt{run()} - most arguments are passed directly through to that interface. However, explicitly passing \texttt{input=None} to inherit the parent's standard input file handle is not supported.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting \texttt{universal_newlines} to \texttt{True} as described above in \textit{Frequently Used Arguments}.

To also capture standard error in the result, use \texttt{stderr=subprocess.STDOUT}:

```
>>> subprocess.check_output(...  "ls non_existent_file; exit 0", ...
...  stderr=subprocess.STDOUT,
...  shell=True)
'ls: non_existent_file: No such file or directory

\textit{New in version 3.1.}

\textit{Changed in version 3.3: timeout was added.}

\textit{Changed in version 3.4: Support for the \texttt{input} keyword argument was added.}

\textit{Changed in version 3.6: encoding and errors were added. See}
run() for details.

New in version 3.7: text was added as a more readable alias for universal_newlines.
Replacing Older Functions with the `subprocess` Module

In this section, “a becomes b” means that b can be used as a replacement for a.

**Note:** All “a” functions in this section fail (more or less) silently if the executed program cannot be found; the “b” replacements raise `OSError` instead.

In addition, the replacements using `check_output()` will fail with a `CalledProcessError` if the requested operation produces a non-zero return code. The output is still available as the `output` attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the `subprocess` module.

**Replacing `/bin/sh` shell backquote**

```bash
output=`mycmd myarg`
```

becomes:

```python
output = check_output(['mycmd', 'myarg'])
```

**Replacing shell pipeline**

```bash
output=`dmesg | grep hda`
```
becomes:

```python
def test_function():
    p1 = Popen(['dmesg'], stdout=PIPE)
    p2 = Popen(['grep', 'hda'], stdin=p1.stdout, stdout=p1.stdout)
    output = p2.communicate()[0]
    p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits before p1.
    output = check_output('dmesg | grep hda', shell=True)
```

The `p1.stdout.close()` call after starting the `p2` is important in order for `p1` to receive a SIGPIPE if `p2` exits before `p1`.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```bash
output=`dmesg | grep hda`
```

becomes:

```python
output=check_output('dmesg | grep hda', shell=True)
```

### Replacing `os.system()`

```python
sts = os.system('mycmd + myarg')
# becomes
sts = call('mycmd + myarg', shell=True)
```

**Notes:**

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```python
try:
    retcode = call('mycmd + myarg', shell=True)
if retcode < 0:
```
```python
print("Child was terminated by signal", -retcode)
else:
    print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

P_NOWAIT example:

```python
pid = os.spawnlp(os.P_NOWAIT, "/bin/mymcd", "mymcd",
    
    pid = Popen(["/bin/mymcd", "myarg"]).pid
```

P_WAIT example:

```python
retcode = os.spawnlp(os.P_WAIT, "/bin/mymcd", "mymcd",
    
    retcode = call(["/bin/mymcd", "myarg"])
```

Vector example:

```python
os.spawnvp(os.P_NOWAIT, path, args)
    
    Popen([path] + args[1:])
```

Environment example:

```python
os.spawnlpe(os.P_NOWAIT, "/bin/mymcd", "mymcd", "myarg",
    
    Popen(["/bin/mymcd", "myarg"], env={"PATH": "/usr/bin"})
```
Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```python
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize=None)
>>>
p = Popen(cmd, shell=True, bufsize=bufsize,
         stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```python
(child_stdin, child_stdout, child_stderr) = os.popen3(cmd, mode, bufsize=None)
>>>
p = Popen(cmd, shell=True, bufsize=bufsize,
         stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin, child_stdout, child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```python
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize=None)
>>>
p = Popen(cmd, shell=True, bufsize=bufsize,
         stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```python
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
```
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")

Replacing functions from the popen2 module

**Note:** If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```python
(child_stdout, child_stdin) = popen2.popen2("somestring")

p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```python
(child_stdout, child_stdin) = popen2.popen2(['"mycmd"

p = Popen(['"mycmd", "myarg"'], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 and popen2.Popen4 basically work as `subprocess.Popen`, except that:

- **Popen** raises an exception if the execution fails.
- The `capturestderr` argument is replaced with the `stderr` argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to
specify `close_fds=True` with `Popen` to guarantee this behavior on all platforms or past Python versions.
Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x commands module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

\texttt{subprocess.getstatusoutput}(cmd)

Return (exitcode, output) of executing \textit{cmd} in a shell.

Execute the string \textit{cmd} in a shell with \texttt{Popen.check_output()} and return a 2-tuple (exitcode, output). The locale encoding is used; see the notes on \texttt{Frequently Used Arguments} for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

\begin{verbatim}
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
\end{verbatim}

\textbf{Availability:} POSIX & Windows.

\textit{Changed in version 3.3.4:} Windows support was added.

The function now returns (exitcode, output) instead of (status,
output) as it did in Python 3.3.3 and earlier. exitcode has the same value as returncode.

subprocess.getoutput(cmd)
Return output (stdout and stderr) of executing cmd in a shell.

Like getstatusoutput(), except the exit code is ignored and the return value is a string containing the command's output. Example:

```python
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: POSIX & Windows.

Changed in version 3.3.4: Windows support added
Notes

Converting an argument sequence to a string on Windows

On Windows, an args sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See also:

shlex
Module which provides function to parse and escape command lines.
The `sched` module defines a class which implements a general purpose event scheduler:

```python
class sched.scheduler(timefunc=time.monotonic, delayfunc=time.sleep)
```

The `scheduler` class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — `timefunc` should be callable without arguments, and return a number (the “time”, in any units whatsoever). If `time.monotonic` is not available, the `timefunc` default is `time.time` instead. The `delayfunc` function should be callable with one argument, compatible with the output of `timefunc`, and should delay that many time units. `delayfunc` will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

*Changed in version 3.3:* `timefunc` and `delayfunc` parameters are optional.

*Changed in version 3.3:* `scheduler` class can be safely used in multi-threaded environments.

Example:

```python
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
```
... print("From print_time", time.time(), a)
...

>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'}
...     s.run()
...     print(time.time())
...

>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
Scheduler Objects

scheduler instances have the following methods and attributes:

scheduler.enterabs(time, priority, action, argument=(), kwargs={})

Schedule a new event. The time argument should be a numeric type compatible with the return value of the timefunc function passed to the constructor. Events scheduled for the same time will be executed in the order of their priority. A lower number represents a higher priority.

Executing the event means executing action(*argument, **kwargs). argument is a sequence holding the positional arguments for action. kwargs is a dictionary holding the keyword arguments for action.

Return value is an event which may be used for later cancellation of the event (see cancel()).

Changed in version 3.3: argument parameter is optional.

New in version 3.3: kwargs parameter was added.

scheduler.enter(delay, priority, action, argument=(), kwargs={})

Schedule an event for delay more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for enterabs().

Changed in version 3.3: argument parameter is optional.

New in version 3.3: kwargs parameter was added.
scheduler.cancel(event)
Remove the event from the queue. If event is not an event currently in the queue, this method will raise a ValueError.

scheduler.empty()
Return true if the event queue is empty.

scheduler.run(blocking=True)
Run all scheduled events. This method will wait (using the delayfunc() function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If blocking is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either action or delayfunc can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by action, the event will not be attempted in future calls to run().

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

*New in version 3.3:* blocking parameter was added.

scheduler.queue
Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a named tuple with the following fields: time, priority, action, argument, kwargs.
The `queue` module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The `Queue` class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the `threading` module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a **FIFO** queue, the first tasks added are the first retrieved. In a **LIFO** queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the `heapq` module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a “simple” **FIFO** queue type, `SimpleQueue`, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The `queue` module defines the following classes and exceptions:

```python
class queue.Queue(maxsize=0)
```
Constructor for a **FIFO** queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

```python
class queue.LifoQueue(maxsize=0)
```
Constructor for a **LIFO** queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

```python
class queue.PriorityQueue(maxsize=0)
```
Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```python
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
```
class 

```python
class PrioritizedItem:
    priority: int
    item: Any = field(compare=False)
```

class queue. 

**SimpleQueue**

Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

*New in version 3.7.*

exception queue. 

**Empty**

Exception raised when non-blocking `get()` (or `get_nowait()`) is called on a `Queue` object which is empty.

exception queue. 

**Full**

Exception raised when non-blocking `put()` (or `put_nowait()`) is called on a `Queue` object which is full.
Queue Objects

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below.

Queue.qsize()
Return the approximate size of the queue. Note, qsize() > 0 doesn’t guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.empty()
Return True if the queue is empty, False otherwise. If empty() returns True it doesn’t guarantee that a subsequent call to put() will not block. Similarly, if empty() returns False it doesn’t guarantee that a subsequent call to get() will not block.

Queue.full()
Return True if the queue is full, False otherwise. If full() returns True it doesn’t guarantee that a subsequent call to get() will not block. Similarly, if full() returns False it doesn’t guarantee that a subsequent call to put() will not block.

Queue.put(item, block=True, timeout=None)
Put item into the queue. If optional args block is true and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Full exception if no free slot was available within that time. Otherwise (block is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (timeout is ignored in that case).
Queue. **put_nowait**(item)
   Equivalent to `put(item, False)`.

Queue. **get**(block=True, timeout=None)
   Remove and return an item from the queue. If optional args `block` is true and `timeout` is None (the default), block if necessary until an item is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the `Empty` exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the `Empty` exception (timeout is ignored in that case).

Queue. **get_nowait**()
   Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue. **task_done**()
   Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

   If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

   Raises a `ValueError` if called more times than there were items placed in the queue.

Queue. **join**()
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Example of how to wait for enqueued tasks to be completed:

```python
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(Nullen)
for t in threads:
```
t.join()
SimpleQueue Objects

SimpleQueue objects provide the public methods described below.

SimpleQueue.qsize()
Return the approximate size of the queue. Note, qsize() > 0 doesn’t guarantee that a subsequent get() will not block.

SimpleQueue.empty()
Return True if the queue is empty, False otherwise. If empty() returns False it doesn’t guarantee that a subsequent call to get() will not block.

SimpleQueue.put(item, block=True, timeout=None)
Put item into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args block and timeout are ignored and only provided for compatibility with Queue.put().

CPython implementation detail: This method has a C implementation which is reentrant. That is, a put() or get() call can be interrupted by another put() call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as __del__ methods or weakref callbacks.

SimpleQueue.put_nowait(item)
Equivalent to put(item), provided for compatibility with Queue.put_nowait().

SimpleQueue.get(block=True, timeout=None)
Remove and return an item from the queue. If optional args block is true and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Empty exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the Empty exception (timeout is ignored in that case).

SimpleQueue.get_nowait()
Equivalent to get(False).

See also:

Class multiprocessing.Queue
A queue class for use in a multi-processing (rather than multi-threading) context.

collections.deque is an alternative implementation of unbounded queues with fast atomic append() and popleft() operations that do not require locking.
_thread — Low-level threading API

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided. The threading module provides an easier to use and higher-level threading API built on top of this module.

*Changed in version 3.7:* This module used to be optional, it is now always available.

This module defines the following constants and functions:

*exception _thread.error*

Raised on thread-specific errors.

*Changed in version 3.3:* This is now a synonym of the built-in RuntimeError.

_*thread.LockType*

This is the type of lock objects.

_*thread.start_new_thread(function, args[, kwargs])*  
Start a new thread and return its identifier. The thread executes the function `function` with the argument list `args` (which must be a tuple). The optional `kwargs` argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a
stack trace is printed and then the thread exits (but other threads continue to run).

_thread.interrupt_main()  
Raise a KeyboardInterrupt exception in the main thread. A subthread can use this function to interrupt the main thread.

_thread.exit()  
Raise the SystemExit exception. When not caught, this will cause the thread to exit silently.

_thread.allocate_lock()  
Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.get_ident()  
Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.stack_size([size])  
Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If size is not specified, 0 is used. If changing the thread stack size is unsupported, a RuntimeError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself.
Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

**Availability:** Windows, systems with POSIX threads.

__thread__ **TIMEOUT_MAX**

The maximum value allowed for the `timeout` parameter of `Lock.acquire()`. Specifying a timeout greater than this value will raise an `OverflowError`.

*New in version 3.2.*

Lock objects have the following methods:

**lock.** **acquire**(waitflag=1, timeout=-1)

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for existence).

If the integer `waitflag` argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as above.

If the floating-point `timeout` argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative `timeout` argument specifies an unbounded wait. You
cannot specify a timeout if waitflag is zero.

The return value is True if the lock is acquired successfully, False if not.

Changed in version 3.2: The timeout parameter is new.

Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

lock.release()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked()

Return the status of the lock: True if it has been acquired by some thread, False if not.

In addition to these methods, lock objects can also be used via the with statement, e.g.:

```python
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Caveats:

- Threads interact strangely with interrupts: the KeyboardInterrupt exception will be received by an arbitrary thread. (When the signal module is available, interrupts always go to the main thread.)
• Calling `sys.exit()` or raising the `SystemExit` exception is equivalent to calling `_thread.exit()`.
• It is not possible to interrupt the `acquire()` method on a lock — the `KeyboardInterrupt` exception will happen after the lock has been acquired.
• When the main thread exits, it is system defined whether the other threads survive. On most systems, they are killed without executing `try ... finally` clauses or executing object destructors.
• When the main thread exits, it does not do any of its usual cleanup (except that `try ... finally` clauses are honored), and the standard I/O files are not flushed.
_dummy_thread — Drop-in replacement for the _thread module

Source code: Lib/_dummy_thread.py

Deprecated since version 3.7: Python now always has threading enabled. Please use _thread (or, better, threading) instead.

This module provides a duplicate interface to the _thread module. It was meant to be imported when the _thread module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
dummy_threading — Drop-in replacement for the threading module

Source code: Lib/dummy_threading.py

Deprecated since version 3.7: Python now always has threading enabled. Please use threading instead.

This module provides a duplicate interface to the threading module. It was meant to be imported when the _thread module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
This module provides APIs to manage, store, and access context-local state. The `ContextVar` class is used to declare and work with Context Variables. The `copy_context()` function and the `Context` class should be used to manage the current context in asynchronous frameworks.

Context managers that have state should use Context Variables instead of `threading.local()` to prevent their state from bleeding to other code unexpectedly, when used in concurrent code.

See also PEP 567 for additional details.

*New in version 3.7.*
Context Variables

`class contextvars.ContextVar(name[, *, default])`

This class is used to declare a new Context Variable, e.g.:

```python
var: ContextVar[int] = ContextVar('var', default=42)
```

The required `name` parameter is used for introspection and debug purposes.

The optional keyword-only `default` parameter is returned by `ContextVar.get()` when no value for the variable is found in the current context.

**Important:** Context Variables should be created at the top module level and never in closures. `Context` objects hold strong references to context variables which prevents context variables from being properly garbage collected.

**name**

The name of the variable. This is a read-only property.

*New in version 3.7.1.*

**get([default])**

Return a value for the context variable for the current context.

If there is no value for the variable in the current context, the method will:

- return the value of the `default` argument of the method, if provided; or
- return the default value for the context variable, if it was created with one; or
- raise a LookupError.

**set**(value)

Call to set a new value for the context variable in the current context.

The required *value* argument is the new value for the context variable.

Returns a *Token* object that can be used to restore the variable to its previous value via the `ContextVar.reset()` method.

**reset**(token)

Reset the context variable to the value it had before the `ContextVar.set()` that created the *token* was used.

For example:

```python
var = ContextVar('var')
token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'
var.reset(token)

# After the reset call the var has no value again
# var.get() would raise a LookupError.
```

**class contextvars.Token**

*Token* objects are returned by the `ContextVar.set()` method. They can be passed to the `ContextVar.reset()` method to revert the value of the variable to what it was before the
corresponding set.

Token. **var**

A read-only property. Points to the `ContextVar` object that created the token.

Token. **old_value**

A read-only property. Set to the value the variable had before the `ContextVar.set()` method call that created the token. It points to `Token.MISSING` is the variable was not set before the call.

Token. **MISSING**

A marker object used by `Token.old_value`. 
Manual Context Management

customvars. **copy_context()**

Returns a copy of the current `Context` object.

The following snippet gets a copy of the current context and prints all variables and their values that are set in it:

```python
ctx: Context = copy_context()
print(list(ctx.items()))
```

The function has an O(1) complexity, i.e. works equally fast for contexts with a few context variables and for contexts that have a lot of them.

customvars. **Context**

A mapping of `ContextVars` to their values.

`Context()` creates an empty context with no values in it. To get a copy of the current context use the `copy_context()` function.

Context implements the `collections.abc.Mapping` interface.

**run**(callable, *args, **kwargs)

Execute `callable(*args, **kwargs)` code in the context object the `run` method is called on. Return the result of the execution or propagate an exception if one occurred.

Any changes to any context variables that `callable` makes will be contained in the context object:

```python
var = ContextVar('var')
var.set('spam')
```
**def** main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' co
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set
# var.get() == 'spam'

The method raises a **RuntimeError** when called on the same context object from more than one OS thread, or when called recursively.

**copy()**

Return a shallow copy of the context object.

**var in context**

Return **True** if the context has a value for var set; return **False** otherwise.
context[var]
Return the value of the var ContextVar variable. If the variable is not set in the context object, a KeyError is raised.

get(var[, default])
Return the value for var if var has the value in the context object. Return default otherwise. If default is not given, return None.

iter(context)
Return an iterator over the variables stored in the context object.

len(proxy)
Return the number of variables set in the context object.

keys()
Return a list of all variables in the context object.

values()
Return a list of all variables' values in the context object.

items()
Return a list of 2-tuples containing all variables and their values in the context object.
asyncio support

Context variables are natively supported in asyncio and are ready to be used without any extra configuration. For example, here is a simple echo server, that uses a context variable to make the address of a remote client available in the Task that handles that client:

```python
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr_var', default=None)

def render_goodbye():
    # The address of the currently handled client can be accessed without passing it explicitly to this function.
    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket')
    client_addr_var.set(addr)

    # In any code that we call is now possible to get client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()
```
```python
async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#       telnet 127.0.0.1 8081
```
Networking and Interprocess Communication

The modules described in this chapter provide mechanisms for networking and inter-processes communication.

Some modules only work for two processes that are on the same machine, e.g. signal and mmap. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

- asyncio — Asynchronous I/O
- socket — Low-level networking interface
- ssl — TLS/SSL wrapper for socket objects
- select — Waiting for I/O completion
- selectors — High-level I/O multiplexing
- asyncore — Asynchronous socket handler
- asynchat — Asynchronous socket command/response handler
- signal — Set handlers for asynchronous events
- mmap — Memory-mapped file support
asyncio — Asynchronous I/O

asyncio is a library to write concurrent code using the async/await syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level structured network code.

asyncio provides a set of high-level APIs to:

- run Python coroutines concurrently and have full control over their execution;
- perform network IO and IPC;
- control subprocesses;
- distribute tasks via queues;
- synchronize concurrent code;

Additionally, there are low-level APIs for library and framework developers to:

- create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc;
implement efficient protocols using transports;
bridge callback-based libraries and code with async/await syntax.

Reference

High-level APIs

- Coroutines and Tasks
- Streams
- Synchronization Primitives
- Subprocesses
- Queues
- Exceptions

Low-level APIs

- Event Loop
- Futures
- Transports and Protocols
- Policies
- Platform Support

Guides and Tutorials

- High-level API Index
- Low-level API Index
- Developing with asyncio
Coroutines and Tasks

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

<table>
<thead>
<tr>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coroutines</td>
</tr>
<tr>
<td>Awaitables</td>
</tr>
<tr>
<td>Running an asyncio Program</td>
</tr>
<tr>
<td>Creating Tasks</td>
</tr>
<tr>
<td>Sleeping</td>
</tr>
<tr>
<td>Running Tasks Concurrently</td>
</tr>
<tr>
<td>Shielding From Cancellation</td>
</tr>
<tr>
<td>Timeouts</td>
</tr>
<tr>
<td>Waiting Primitives</td>
</tr>
<tr>
<td>Scheduling From Other Threads</td>
</tr>
<tr>
<td>Introspection</td>
</tr>
<tr>
<td>Task Object</td>
</tr>
<tr>
<td>Generator-based Coroutines</td>
</tr>
</tbody>
</table>
Coroutines

Coroutines declared with async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code (requires Python 3.7+) prints “hello”, waits 1 second, and then prints “world”:

```python
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```python
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine asyncio provides three main mechanisms:

- The `asyncio.run()` function to run the top-level entry point “main()” function (see the above example.)

- Awaiting on a coroutine. The following snippet of code will print “hello” after waiting for 1 second, and then print “world” after waiting for another 2 seconds:

```python
import asyncio
import time
```
async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at \{time.strftime('%X')\}"")
    await say_after(1, 'hello')
    await say_after(2, 'world')
    print(f"finished at \{time.strftime('%X')\}")

asyncio.run(main())

Expected output:

started at 17:13:52
hello
world
finished at 17:13:55

- The asyncio.create_task() function to run coroutines concurrently as asyncio Tasks.

Let's modify the above example and run two say_after coroutines concurrently:

async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at \{time.strftime('%X')\}"
```python
# Wait until both tasks are completed (should # around 2 seconds.)
await task1
await task2

print(f"finished at {time.strftime('%X')}\")
```

Note that expected output now shows that the snippet runs 1 second faster than before:

```
started at 17:14:32
hello
world
finished at 17:14:34
```
Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of **awaitable** objects: **coroutines**, **Tasks**, and **Futures**.

**Coroutines**

Python coroutines are **awaitables** and therefore can be awaited from other coroutines:

```python
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited, 
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested()) # will print "42".

asyncio.run(main())
```

**Important**: In this documentation the term “coroutine” can be used for two closely related concepts:
• a coroutine function: an async def function;
• a coroutine object: an object returned by calling a coroutine function.

asyncio also supports legacy generator-based coroutines.

Tasks

*Tasks* are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like asyncio.create_task() the coroutine is automatically scheduled to run soon:

```python
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete
    await task

asyncio.run(main())
```

Futures

A *Future* is a special low-level awaitable object that represents an eventual result of an asynchronous operation.
When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```python
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is `loop.run_in_executor()`. 
Running an asyncio Program

asyncio.run(coro, *, debug=False)

This function runs the passed coroutine, taking care of managing the asyncio event loop and finalizing asynchronous generators.

This function cannot be called when another asyncio event loop is running in the same thread.

If debug is True, the event loop will be run in debug mode.

This function always creates a new event loop and closes it at the end. It should be used as a main entry point for asyncio programs, and should ideally only be called once.

New in version 3.7: Important: this function has been added to asyncio in Python 3.7 on a provisional basis.
Creating Tasks

\texttt{asyncio.create_task(coro)}

Wrap the \texttt{coro} coroutine into a \texttt{Task} and schedule its execution. Return the Task object.

The task is executed in the loop returned by \texttt{get_running_loop()}, \texttt{RuntimeError} is raised if there is no running loop in current thread.

This function has been \textbf{added in Python 3.7}. Prior to Python 3.7, the low-level \texttt{asyncio.ensure_future()} function can be used instead:

\begin{verbatim}
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable:
task = asyncio.ensure_future(coro())
...
\end{verbatim}

\textit{New in version 3.7.}
Sleeping

coroutine asyncio.sleep(delay, result=None, *, loop=None)

Block for delay seconds.

If result is provided, it is returned to the caller when the coroutine completes.

sleep() always suspends the current task, allowing other tasks to run.

The loop argument is deprecated and scheduled for removal in Python 3.10.

Example of coroutine displaying the current date every second for 5 seconds:

```python
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```
Running Tasks Concurrently

```
awaitable asyncio.gather(*aws, loop=None, return_exceptions=False)
```

Run **awaitable objects** in the **aws** sequence *concurrently*.

If any awaitable in **aws** is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in **aws**.

If **return_exceptions** is **False** (default), the first raised exception is immediately propagated to the task that awaits on **gather()**. Other awaitables in the **aws** sequence **won’t be cancelled** and will continue to run.

If **return_exceptions** is **True**, exceptions are treated the same as successful results, and aggregated in the result list.

If **gather()** is **cancelled**, all submitted awaitables (that have not completed yet) are also **cancelled**.

If any Task or Future from the **aws** sequence is **cancelled**, it is treated as if it raised **CancelledError** – the **gather()** call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

Example:

```
import asyncio
```
async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
# Task A: Compute factorial(2)...
# Task B: Compute factorial(2)...
# Task C: Compute factorial(2)...
# Task A: factorial(2) = 2
# Task B: Compute factorial(3)...
# Task C: Compute factorial(3)...
# Task B: factorial(3) = 6
# Task C: Compute factorial(4)...
# Task C: factorial(4) = 24

Changed in version 3.7: If the gather itself is cancelled, the cancellation is propagated regardless of return_exceptions.
Shielding From Cancellation

`awaitable asyncio.shield(aw, *, loop=None)`

Protect an `awaitable object` from being `cancelled`.

If `aw` is a coroutine it is automatically scheduled as a Task.

The statement:

```python
res = await shield(something())
```

is equivalent to:

```python
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the “await” expression still raises a `CancelledError`.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a `try/except` clause, as follows:

```python
try:
    res = await shield(something())
except CancelledError:
    res = None
```
Timeouts

coroutine asyncio.wait_for(aw, timeout, *, loop=None)

Wait for the aw awaitable to complete with a timeout.

If aw is a coroutine it is automatically scheduled as a Task.

timeout can either be None or a float or int number of seconds to wait for. If timeout is None, block until the future completes.

If a timeout occurs, it cancels the task and raises asyncio.TimeoutError.

To avoid the task cancellation, wrap it in shield().

The function will wait until the future is actually cancelled, so the total wait time may exceed the timeout.

If the wait is cancelled, the future aw is also cancelled.

The loop argument is deprecated and scheduled for removal in Python 3.10.

Example:

```python
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1)
    except asyncio.TimeoutError:
        print('Time out!')
```

```python
except asyncio.TimeoutError:
    print('timeout!')

asyncio.run(main())

# Expected output:
#
# timeout!
```

*Changed in version 3.7:* When `aw` is cancelled due to a timeout, `wait_for` waits for `aw` to be cancelled. Previously, it raised `asyncio.TimeoutError` immediately.
Waiting Primitives

`coroutine asyncio.wait(aws, *, loop=None, timeout=None, return_when=ALL_COMPLETED)`

Run awaitable objects in the `aws` set concurrently and block until the condition specified by `return_when`.

If any awaitable in `aws` is a coroutine, it is automatically scheduled as a Task. Passing coroutines objects to `wait()` directly is deprecated as it leads to confusing behavior.

Returns two sets of Tasks/Futures: `(done, pending)`.

Usage:

```python
done, pending = await asyncio.wait(aws)
```

The `loop` argument is deprecated and scheduled for removal in Python 3.10.

`timeout` (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise `asyncio.TimeoutError`. Futures or Tasks that aren’t done when the timeout occurs are simply returned in the second set.

`return_when` indicates when this function should return. It must be one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST_COMPLETED</td>
<td>The function will return when any future finishes or is cancelled.</td>
</tr>
<tr>
<td>FIRST_EXCEPTION</td>
<td>The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ALL_COMPLETED</td>
<td>The function will return when all futures finish or are cancelled.</td>
</tr>
</tbody>
</table>

Unlike `wait_for()`, `wait()` does not cancel the futures when a timeout occurs.

**Note:** `wait()` schedules coroutines as Tasks automatically and later returns those implicitly created Task objects in `(done, pending)` sets. Therefore the following code won’t work as expected:

```python
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

Here is how the above snippet can be fixed:

```python
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```
Passing coroutine objects to `wait()` directly is deprecated.

```python
asyncio.as_completed(aws, *, loop=None, timeout=None)
```

Run `awaitable objects` in the `aws` set concurrently. Return an iterator of `Future` objects. Each Future object returned represents the earliest result from the set of the remaining awaitables.

Raises `asyncio.TimeoutError` if the timeout occurs before all Futures are done.

Example:

```python
for f in asyncio.as_completed(aws):
    earliest_result = await f
    # ...
```
Scheduling From Other Threads

`asyncio.run_coroutine_threadsafe(coro, loop)`
Submit a coroutine to the given event loop. Thread-safe.

Return a `concurrent.futures.Future` to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```python
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```python
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc}')
else:
    print(f'The coroutine returned: {result!r}')
```
See the `concurrency and multithreading` section of the documentation.

Unlike other asyncio functions this function requires the `loop` argument to be passed explicitly.

*New in version 3.5.1.*
**Introspection**

`asyncio.current_task(loop=None)`

Return the currently running `Task` instance, or `None` if no task is running.

If `loop` is `None`, `get_running_loop()` is used to get the current loop.

*New in version 3.7.*

`asyncio.all_tasks(loop=None)`

Return a set of not yet finished `Task` objects run by the loop.

If `loop` is `None`, `get_running_loop()` is used for getting current loop.

*New in version 3.7.*
**Task Object**

```python
class asyncio.Task(coro, *, loop=None)
```


Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level `asyncio.create_task()` function to create Tasks, or the low-level `loop.create_task()` or `ensure_future()` functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the `cancel()` method. Calling it will cause the Task to throw a `CancelledError` exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

`cancelled()` can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the `CancelledError` exception and was actually cancelled.

`asyncio.Task` inherits from `Future` all of its APIs except
Future.set_result() and Future.set_exception().

Tasks support the contextvars module. When a Task is created it copies the current context and later runs its coroutine in the copied context.

Changed in version 3.7: Added support for the contextvars module.

cancel()

Request the Task to be cancelled.

This arranges for a CancelledError exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a try ... ... except CancelledError ... finally block. Therefore, unlike Future.cancel(), Task.cancel() does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged.

The following example illustrates how coroutines can intercept the cancellation request:

```python
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
```
```python
raise

finally:
    print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

done()
```

cancelled()

Return True if the Task is cancelled.

The Task is cancelled when the cancellation was requested with cancel() and the wrapped coroutine propagated the CancelledError exception thrown into it.

done()

Return True if the Task is done.
A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

**result()**

Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

If the Task has been *cancelled*, this method raises a `CancelledError` exception.

If the Task’s result isn’t yet available, this method raises a `InvalidStateError` exception.

**exception()**

Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

If the Task has been *cancelled*, this method raises a `CancelledError` exception.

If the Task isn’t *done* yet, this method raises an `InvalidStateError` exception.

**add_done_callback**(callback, *, context=None)

Add a callback to be run when the Task is *done*.

This method should only be used in low-level callback-based code.
See the documentation of `Future.add_done_callback()` for more details.

**remove_done_callback(callback)**
Remove `callback` from the callbacks list.

This method should only be used in low-level callback-based code.

See the documentation of `Future.remove_done_callback()` for more details.

**get_stack(***, *limit=None)**
Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional `limit` argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

**print_stack(***, *limit=None, file=None)**
Print the stack or traceback for this Task.
This produces output similar to that of the traceback module for the frames retrieved by `get_stack()`.

The `limit` argument is passed to `get_stack()` directly.

The `file` argument is an I/O stream to which the output is written; by default output is written to `sys.stderr`.

**classmethod all_tasks**(loop=None)
Return a set of all tasks for an event loop.

By default all tasks for the current event loop are returned. If `loop` is None, the `get_event_loop()` function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the `asyncio.all_tasks()` function instead.

**classmethod current_task**(loop=None)
Return the currently running task or None.

If `loop` is None, the `get_event_loop()` function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the `asyncio.current_task()` function instead.
Generator-based Coroutines

**Note:** Support for generator-based coroutines is deprecated and is scheduled for removal in Python 3.10.

Generator-based coroutines predate async/await syntax. They are Python generators that use `yield from` expressions to await on Futures and other coroutines.

Generator-based coroutines should be decorated with `@asyncio.coroutine`, although this is not enforced.

```python
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

This decorator is deprecated and is scheduled for removal in Python 3.10.

This decorator should not be used for `async def` coroutines.

```python
asyncio.iscoroutine(obj)
```

Return `True` if `obj` is a coroutine object.
This method is different from `inspect.iscoroutine()` because it returns `True` for generator-based coroutines.

```python
asyncio.iscoroutinefunction(func)
```

Return `True` if `func` is a coroutine function.

This method is different from `inspect.iscoroutinefunction()` because it returns `True` for generator-based coroutine functions decorated with `@coroutine`.
Streams

Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.

Here is an example of a TCP echo client written using asyncio streams:

```python
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection('127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also the Examples section below.

Stream Functions

The following top-level asyncio functions can be used to create and work with streams:
coroutine asyncio.open_connection(host=None, port=None, *, loop=None, limit=None, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None)

Establish a network connection and return a pair of (reader, writer) objects.

The returned reader and writer objects are instances of StreamReader and StreamWriter classes.

The loop argument is optional and can always be determined automatically when this function is awaited from a coroutine.

limit determines the buffer size limit used by the returned StreamReader instance. By default the limit is set to 64 KiB.

The rest of the arguments are passed directly to loop.create_connection().

New in version 3.7: The ssl_handshake_timeout parameter.

coroutine asyncio.start_server(client_connected_cb, host=None, port=None, *, loop=None, limit=None, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, ssl_handshake_timeout=None, start_serving=True)

Start a socket server.

The client_connected_cb callback is called whenever a new client connection is established. It receives a (reader, writer) pair as two arguments, instances of the StreamReader and StreamWriter classes.

client_connected_cb can be a plain callable or a coroutine function;
if it is a coroutine function, it will be automatically scheduled as a 
Task.

The loop argument is optional and can always be determined 
automatically when this method is awaited from a coroutine.

limit determines the buffer size limit used by the returned 
StreamReader instance. By default the limit is set to 64 KiB.

The rest of the arguments are passed directly to 
loop.create_server().

New in version 3.7: The ssl_handshake_timeout and start_serving 
parameters.

Unix Sockets

coroutine asyncio.open_unix_connection(path=None, *, 
  loop=None, limit=None, ssl=None, sock=None, 
  server_hostname=None, ssl_handshake_timeout=None) 

Establish a Unix socket connection and return a pair of (reader,
writer).

Similar to open_connection() but operates on Unix sockets.

See also the documentation of 
loop.create_unix_connection().

Availability: Unix.

New in version 3.7: The ssl_handshake_timeout parameter.

Changed in version 3.7: The path parameter can now be a path-
like object
coroutine asyncio.\texttt{start_unix_server}(\texttt{client\_connected\_cb}, \texttt{path}=None, *, \texttt{loop}=None, \texttt{limit}=None, \texttt{sock}=None, \texttt{backlog}=100, \texttt{ssl}=None, \texttt{ssl\_handshake\_timeout}=None, \texttt{start\_serving}=True)  
Start a Unix socket server.

Similar to \texttt{start\_server() } but works with Unix sockets.

See also the documentation of \texttt{loop\_create\_unix\_server()}.  

\textbf{Availability:} Unix.

\textit{New in version 3.7:} The \texttt{ssl\_handshake\_timeout} and \texttt{start\_serving} parameters.

\textit{Changed in version 3.7:} The \texttt{path} parameter can now be a path-like object.
StreamReader

class asyncio.StreamReader

Represents a reader object that provides APIs to read data from the IO stream.

It is not recommended to instantiate StreamReader objects directly; use open_connection() and start_server() instead.

coroutine read(n=-1)

Read up to n bytes. If n is not provided, or set to -1, read until EOF and return all read bytes.

If EOF was received and the internal buffer is empty, return an empty bytes object.

coroutine readline()

Read one line, where “line” is a sequence of bytes ending with \n.

If EOF is received and \n was not found, the method returns partially read data.

If EOF is received and the internal buffer is empty, return an empty bytes object.

coroutine readexactly(n)

Read exactly n bytes.

Raise an IncompleteReadError if EOF is reached before n can be read. Use the IncompleteReadError.partial attribute to get the partially read data.
Coroutine `readuntil(separator=b'\n')`

Read data from the stream until `separator` is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

If the amount of data read exceeds the configured stream limit, a `LimitOverrunError` exception is raised, and the data is left in the internal buffer and can be read again.

If EOF is reached before the complete separator is found, an `IncompleteReadError` exception is raised, and the internal buffer is reset. The `IncompleteReadError.partial` attribute may contain a portion of the separator.

*New in version 3.5.2.*

`at_eof()`

Return `True` if the buffer is empty and `feed_eof()` was called.
StreamWriter

class asyncio.StreamWriter

Represents a writer object that provides APIs to write data to the IO stream.

It is not recommended to instantiate StreamWriter objects directly; use open_connection() and start_server() instead.

can_write_eof()

Return True if the underlying transport supports the write_eof() method, False otherwise.

write_eof()

Close the write end of the stream after the buffered write data is flushed.

transport

Return the underlying asyncio transport.

get_extra_info(name, default=None)

Access optional transport information; see BaseTransport.get_extra_info() for details.

write(data)

Write data to the stream.

This method is not subject to flow control. Calls to write() should be followed by drain().

writelines(data)

Write a list (or any iterable) of bytes to the stream.
This method is not subject to flow control. Calls to `writelines()` should be followed by `drain()`.

**coroutine `drain()`**

Wait until it is appropriate to resume writing to the stream. Example:

```python
writer.write(data)
await writer.drain()
```

This is a flow control method that interacts with the underlying IO write buffer. When the size of the buffer reaches the high watermark, `drain()` blocks until the size of the buffer is drained down to the low watermark and writing can be resumed. When there is nothing to wait for, the `drain()` returns immediately.

**close()**

Close the stream.

**is_closing()**

Return `True` if the stream is closed or in the process of being closed.

*New in version 3.7.*

**coroutine `wait_closed()`**

Wait until the stream is closed.

Should be called after `close()` to wait until the underlying connection is closed.

*New in version 3.7.*
Examples

TCP echo client using streams

TCP echo client using the `asyncio.open_connection()` function:

```python
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection('127.0.0.1', 8888)
    print(f'Send: {message!r}')
    writer.write(message.encode())
    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')
    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also: The TCP echo client protocol example uses the low-level `loop.create_connection()` method.

TCP echo server using streams

TCP echo server using the `asyncio.start_server()` function:

```python
import asyncio

async def handle_echo(reader, writer):
    message = await reader.read(100)
    print(f'Received: {message!r}')
    writer.write(message)
    await writer.drain()
```

```python
asyncio.get_event_loop().create_server(handleEcho, '127.0.0.1', 8888)
```
async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())

See also: The TCP echo server protocol example uses the loop.create_server() method.

Get HTTP headers

Simple example querying HTTP headers of the URL passed on the command line:

```python
import asyncio
import urllib.parse
```
```python
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(url.hostname, 80)

    query = ('HEAD {url.path or '/'} HTTP/1.0\r\n' +
             'Host: {url.hostname}\r\n' +
             '\r\n')

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header: {line}')

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Usage:
```
python example.py http://example.com/path/page.html
```
or with HTTPS:

```
python example.py https://example.com/path/page.html
```

Register an open socket to wait for data using streams

Coroutine waiting until a socket receives data using the `open_connection()` function:

```python
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(s

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
```
# Close the second socket

```
wssock.close()
```

```
asyncio.run(wait_for_data())
```

---

**See also:** The register an open socket to wait for data using a protocol example uses a low-level protocol and the `loop.create_connection()` method.

The watch a file descriptor for read events example uses the low-level `loop.add_reader()` method to watch a file descriptor.
Synchronization Primitives

asyncio synchronization primitives are designed to be similar to those of the threading module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use threading for that);
- methods of these synchronization primitives do not accept the timeout argument; use the asyncio.wait_for() function to perform operations with timeouts.

asyncio has the following basic synchronization primitives:

- Lock
- Event
- Condition
- Semaphore
- BoundedSemaphore
Lock

class asyncio.Lock(*, loop=None)

`Lock` implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```python
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```python
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

coroutine `acquire()`

Acquire the lock.

This method waits until the lock is `unlocked`, sets it to `locked` and returns `True`.

`release()`
Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a `RuntimeError` is raised.

**locked()**

Return `True` if the lock is *locked*. 
Event

class asyncio.Event(*, loop=None)
An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is set to true. The flag is set to false initially.

Example:

```python
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task
```
asyncio.run(main())

Coroutine `wait()`

Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls `set()`.

`set()`

Set the event.

All tasks waiting for event to be set will be immediately awakened.

`clear()`

Clear (unset) the event.

Tasks awaiting on `wait()` will now block until the `set()` method is called again.

`is_set()`

Return `True` if the event is set.
**Condition**

`class asyncio.Condition(lock=None, *, loop=None)`


An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an `Event` and a `Lock`. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional `lock` argument must be a `Lock` object or `None`. In the latter case a new Lock object is created automatically.

The preferred way to use a Condition is an `async with` statement:

```python
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```python
cond = asyncio.Condition()

# ... later
await lock.acquire()
```
try:
    await cond.wait()
finally:
    lock.release()

coroutine `acquire()`

Acquire the underlying lock.

This method waits until the underlying lock is `unlocked`, sets it to `locked` and returns `True`.

`notify(n=1)`

Wake up at most `n` tasks (1 by default) waiting on this condition. The method is no-op if no tasks are waiting.

The lock must be acquired before this method is called and released shortly after. If called with an `unlocked` lock a `RuntimeError` error is raised.

`locked()`

Return `True` if the underlying lock is acquired.

`notify_all()`

Wake up all tasks waiting on this condition.

This method acts like `notify()`, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an `unlocked` lock a `RuntimeError` error is raised.

`release()`

Release the underlying lock.
When invoked on an unlocked lock, a `RuntimeError` is raised.

**coroutine** `wait()`

Wait until notified.

If the calling task has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call. Once awakened, the Condition re-acquires its lock and this method returns `True`.

**coroutine** `wait_for(predicate)`

Wait until a predicate becomes `true`.

The predicate must be a callable which result will be interpreted as a boolean value. The final value is the return value.
Semaphore

class asyncio.Semaphore(value=1, *, loop=None)

A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some task calls `release()`.

The optional `value` argument gives the initial value for the internal counter (1 by default). If the given value is less than 0 a `ValueError` is raised.

The preferred way to use a Semaphore is an `async with` statement:

```python
sem = asyncio.Semaphore(10)
# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```python
sem = asyncio.Semaphore(10)
# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```
**coroutine** `acquire()`

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return `True` immediately. If it is zero, wait until a `release()` is called and return `True`.

**locked()**

Returns `True` if semaphore can not be acquired immediately.

**release()**

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike `BoundedSemaphore`, `Semaphore` allows making more `release()` calls than `acquire()` calls.
BoundedSemaphore

```
class asyncio.BoundedSemaphore(value=1, *, loop=None)
```

A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of Semaphore that raises a ValueError in `release()` if it increases the internal counter above the initial `value`.

*Deprecated since version 3.7:* Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (with `await lock`, with (yield from lock)) is deprecated. Use `async with` lock instead.
Subprocesses

This section describes high-level async/await asyncio APIs to create and manage subprocesses.

Here's an example of how asyncio can run a shell command and obtain its result:

```python
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]
{stdout.decode()}')
    if stderr:
        print(f'[stderr]
{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

will print:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Because all asyncio subprocess functions are asynchronous and
asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel. It is indeed trivial to modify the above example to run several commands simultaneously:

```python
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

See also the Examples subsection.
Creating Subprocesses

coroutine asyncio.create_subprocess_exec(*args, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds)

Create a subprocess.

The limit argument sets the buffer limit for StreamReader wrappers for Process.stdout and Process.stderr (if subprocess.PIPE is passed to stdout and stderr arguments).

Return a Process instance.

See the documentation of loop.subprocess_exec() for other parameters.

coroutine asyncio.create_subprocess_shell(cmd, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds)

Run the cmd shell command.

The limit argument sets the buffer limit for StreamReader wrappers for Process.stdout and Process.stderr (if subprocess.PIPE is passed to stdout and stderr arguments).

Return a Process instance.

See the documentation of loop.subprocess_shell() for other parameters.

Important: It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid
shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special shell characters in strings that are going to be used to construct shell commands.

**Note:** The default asyncio event loop implementation on Windows does not support subprocesses. Subprocesses are available for Windows if a `ProactorEventLoop` is used. See Subprocess Support on Windows for details.

**See also:** asyncio also has the following low-level APIs to work with subprocesses: `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, as well as the Subprocess Transports and Subprocess Protocols.
Constants

`asyncio.subprocess.PIPE`

Can be passed to the `stdin`, `stdout` or `stderr` parameters.

If `PIPE` is passed to `stdin` argument, the `Process.stdin` attribute will point to a `StreamWriter` instance.

If `PIPE` is passed to `stdout` or `stderr` arguments, the `Process.stdout` and `Process.stderr` attributes will point to `StreamReader` instances.

`asyncio.subprocess.STDOUT`

Special value that can be used as the `stderr` argument and indicates that standard error should be redirected into standard output.

`asyncio.subprocess.DEVNULL`

Special value that can be used as the `stdin`, `stdout` or `stderr` argument to process creation functions. It indicates that the special file `os.devnull` will be used for the corresponding subprocess stream.
Interacting with Subprocesses

Both `create_subprocess_exec()` and `create_subprocess_shell()` functions return instances of the `Process` class. `Process` is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

`class asyncio.subprocess.Process`

An object that wraps OS processes created by the `create_subprocess_exec()` and `create_subprocess_shell()` functions.

This class is designed to have a similar API to the `subprocess.Popen` class, but there are some notable differences:

- unlike Popen, Process instances do not have an equivalent to the `poll()` method;
- the `communicate()` and `wait()` methods don't have a `timeout` parameter: use the `wait_for()` function;
- the `Process.wait()` method is asynchronous, whereas `subprocess.Popen.wait()` method is implemented as a blocking busy loop;
- the `universal_newlines` parameter is not supported.

This class is not thread safe.

See also the Subprocess and Threads section.

`coroutine wait()`

Wait for the child process to terminate.

Set and return the `returncode` attribute.
Note: This method can deadlock when using stdout=PIPE or stderr=PIPE and the child process generates so much output that it blocks waiting for the OS pipe buffer to accept more data. Use the `communicate()` method when using pipes to avoid this condition.

coroutine **communicate**(input=None)

Interact with process:

1. send data to stdin (if input is not None);
2. read data from stdout and stderr, until EOF is reached;
3. wait for process to terminate.

The optional input argument is the data (bytes object) that will be sent to the child process.

Return a tuple `(stdout_data, stderr_data)`.

If either `BrokenPipeError` or `ConnectionResetError` exception is raised when writing input into stdin, the exception is ignored. This condition occurs when the process exits before all data are written into stdin.

If it is desired to send data to the process' stdin, the process needs to be created with stdin=PIPE. Similarly, to get anything other than None in the result tuple, the process has to be created with stdout=PIPE and/or stderr=PIPE arguments.

Note, that the data read is buffered in memory, so do not use this method if the data size is large or unlimited.

**send_signal**(signal)

Sends the signal `signal` to the child process.
Note: On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a `creationflags` parameter which includes CREATE_NEW_PROCESS_GROUP.

```
Note: On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a `creationflags` parameter which includes CREATE_NEW_PROCESS_GROUP.

```

**terminate()**
Stop the child process.

On POSIX systems this method sends `signal.SIGTERM` to the child process.

On Windows the Win32 API function `TerminateProcess()` is called to stop the child process.

**kill()**
Kill the child.

On POSIX systems this method sends `SIGKILL` to the child process.

On Windows this method is an alias for `terminate()`.

**stdin**
Standard input stream (`StreamWriter`) or `None` if the process was created with `stdin=None`.

**stdout**
Standard output stream (`StreamReader`) or `None` if the process was created with `stdout=None`.

**stderr**
Standard error stream (`StreamReader`) or `None` if the process
was created with stderr=None.

**Warning:** Use the `communicate()` method rather than `process.stdin.write()`, `await process.stdout.read()` or `await process.stderr.read`. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

**pid**

Process identification number (PID).

Note that for processes created by the `create_subprocess_shell()` function, this attribute is the PID of the spawned shell.

**returncode**

Return code of the process when it exits.

A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

### Subprocess and Threads

Standard asyncio event loop supports running subprocesses from different threads, but there are limitations:

- An event loop must run in the main thread.
- The child watcher must be instantiated in the main thread before executing subprocesses from other threads. Call the `get_child_watcher()` function in the main thread to instantiate the child watcher.
Note that alternative event loop implementations might not share the above limitations; please refer to their documentation.

See also: The Concurrency and multithreading in asyncio section.

Examples

An example using the Process class to control a subprocess and the StreamReader class to read from its standard output.

The subprocess is created by the create_subprocess_exec() function:

```python
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())

    # Create the subprocess; redirect the standard output into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
```
asyncio.WindowsProactorEventLoopPolicy()

date = asyncio.run(get_date())
print(f"Current date: {date}")

See also the **same example** written using low-level APIs.
Queues

asyncio queues are designed to be similar to classes of the queue module. Although asyncio queues are not thread-safe, they are designed to be used specifically in async/await code.

Note that methods of asyncio queues don’t have a timeout parameter; use asyncio.wait_for() function to do queue operations with a timeout.

See also the Examples section below.
Queue

```
class asyncio.Queue(maxsize=0, *, loop=None)
```

A first in, first out (FIFO) queue.

If `maxsize` is less than or equal to zero, the queue size is infinite. If it is an integer greater than 0, then `await put()` blocks when the queue reaches `maxsize` until an item is removed by `get()`.

Unlike the standard library threading `queue`, the size of the queue is always known and can be returned by calling the `qsize()` method.

This class is **not thread safe**.

**maxsize**

Number of items allowed in the queue.

**empty()**

Return `True` if the queue is empty, `False` otherwise.

**full()**

Return `True` if there are `maxsize` items in the queue.

If the queue was initialized with `maxsize=0` (the default), then `full()` never returns `True`.

**coroutine get()**

Remove and return an item from the queue. If queue is empty, wait until an item is available.

**get_nowait()**
Return an item if one is immediately available, else raise QueueEmpty.

**coroutine join()**
Block until all items in the queue have been received and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer coroutine calls task_done() to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, join() unblocks.

**coroutine put(item)**
Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.

**put_nowait(item)**
Put an item into the queue without blocking.

If no free slot is immediately available, raise QueueFull.

**qsize()**
Return the number of items in the queue.

**task_done()**
Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items
have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises `ValueError` if called more times than there were items placed in the queue.
Priority Queue

class asyncio.PriorityQueue

A variant of Queue; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form (priority_number, data).
LIFO Queue

class asyncio.LifoQueue

A variant of Queue that retrieves most recently added entries first (last in, first out).
Exceptions

(exception asyncio.QueueEmpty)
This exception is raised when the get_nowait() method is called on an empty queue.

(exception asyncio.QueueFull)
Exception raised when the put_nowait() method is called on a queue that has reached its maxsize.
Examples

Queues can be used to distribute workload between several concurrent tasks:

```python
import asyncio
import random
import time

async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f}')

async def main():
    # Create a queue that we will use to store our 'workload'.
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)
```

# Create three worker tasks to process the queue concurrently.
tasks = []
for i in range(3):
    task = asyncio.create_task(worker(f'worker-{i}'))
    tasks.append(task)

# Wait until the queue is fully processed.
started_at = time.monotonic()
await queue.join()
total_slept_for = time.monotonic() - started_at

# Cancel our worker tasks.
for task in tasks:
    task.cancel()

# Wait until all worker tasks are cancelled.
await asyncio.gather(*tasks, return_exceptions=True)

print('=====
print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
print(f'total expected sleep time: {total_sleep_time:.2f} seconds')

asyncio.run(main())
Exceptions

`exception asyncio.TimeoutError`

The operation has exceeded the given deadline.

**Important:** This exception is different from the builtin `TimeoutError` exception.

`exception asyncio.CancelledError`

The operation has been cancelled.

This exception can be caught to perform custom operations when asyncio Tasks are cancelled. In almost all situations the exception must be re-raised.

**Important:** This exception is a subclass of `Exception`, so it can be accidentally suppressed by an overly broad `try..except` block:

```python
try:
    await operation
except Exception:
    # The cancellation is broken because the *except* block suppresses the CancelledError exception.
    log.log('an error has occurred')
```

Instead, the following pattern should be used:

```python
try:
    await operation
except asyncio.CancelledError:
    raise
```
```python
except Exception:
    log.log('an error has occurred')
```

*exception* asyncio. **InvalidStateError**
Invalid internal state of Task or Future.

Can be raised in situations like setting a result value for a Future object that already has a result value set.

*exception* asyncio. **SendfileNotAvailableError**
The “sendfile” syscall is not available for the given socket or file type.

A subclass of RuntimeError.

*exception* asyncio. **IncompleteReadError**
The requested read operation did not complete fully.

Raised by the asyncio stream APIs.

This exception is a subclass of EOFError.

**expected**
The total number (int) of expected bytes.

**partial**
A string of bytes read before the end of stream was reached.

*exception* asyncio. **LimitOverrunError**
Reached the buffer size limit while looking for a separator.

Raised by the asyncio stream APIs.

**consumed**
The total number of to be consumed bytes.
Event Loop

Preface

The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

Application developers should typically use the high-level asyncio functions, such as `asyncio.run()`, and should rarely need to reference the loop object or call its methods. This section is intended mostly for authors of lower-level code, libraries, and frameworks, who need finer control over the event loop behavior.

Obtaining the Event Loop

The following low-level functions can be used to get, set, or create an event loop:

`asyncio.get_running_loop()`
Return the running event loop in the current OS thread.

If there is no running event loop a `RuntimeError` is raised. This function can only be called from a coroutine or a callback.

*New in version 3.7.*

`asyncio.get_event_loop()`
Get the current event loop. If there is no current event loop set in the current OS thread and `set_event_loop()` has not yet been called, asyncio will create a new event loop and set it as the current one.
Because this function has rather complex behavior (especially when custom event loop policies are in use), using the `get_running_loop()` function is preferred to `get_event_loop()` in coroutines and callbacks.

Consider also using the `asyncio.run()` function instead of using lower level functions to manually create and close an event loop.

```python
asyncio.set_event_loop(loop)
```
Set `loop` as a current event loop for the current OS thread.

```python
asyncio.new_event_loop()
```
Create a new event loop object.

Note that the behaviour of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()` functions can be altered by setting a custom event loop policy.

**Contents**

This documentation page contains the following sections:

- The **Event Loop Methods** section is the reference documentation of the event loop APIs;
- The **Callback Handles** section documents the `Handle` and `TimerHandle` instances which are returned from scheduling methods such as `loop.call_soon()` and `loop.call_later()`;
- The **Server Objects** section documents types returned from event loop methods like `loop.create_server()`;
- The **Event Loop Implementations** section documents the `SelectorEventLoop` and `ProactorEventLoop` classes;
The **Examples** section showcases how to work with some event loop APIs.
Event Loop Methods

Event loops have low-level APIs for the following:

- Running and stopping the loop
- Scheduling callbacks
- Scheduling delayed callbacks
- Creating Futures and Tasks
- Opening network connections
- Creating network servers
- Transferring files
- TLS Upgrade
- Watching file descriptors
- Working with socket objects directly
- DNS
- Working with pipes
- Unix signals
- Executing code in thread or process pools
- Error Handling API
- Enabling debug mode
- Running Subprocesses

Running and stopping the loop

loop.run_until_complete(future)

Run until the future (an instance of Future) has completed.

If the argument is a coroutine object it is implicitly scheduled to run as a asyncio.Task.

Return the Future's result or raise its exception.
loop.run_forever()
Run the event loop until stop() is called.

If stop() is called before run_forever() is called, the loop will poll the I/O selector once with a timeout of zero, run all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exit.

If stop() is called while run_forever() is running, the loop will run the current batch of callbacks and then exit. Note that new callbacks scheduled by callbacks will not run in this case; instead, they will run the next time run_forever() or run_until_complete() is called.

loop.stop()
Stop the event loop.

loop.is_running()
Return True if the event loop is currently running.

loop.is_closed()
Return True if the event loop was closed.

loop.close()
Close the event loop.

The loop must not be running when this function is called. Any pending callbacks will be discarded.

This method clears all queues and shuts down the executor, but does not wait for the executor to finish.

This method is idempotent and irreversible. No other methods
should be called after the event loop is closed.

coroutine loop. **shutdown_asyncgens()**
Schedule all currently open **asynchronous generator** objects to close with an **aclose()** call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

Note that there is no need to call this function when **asyncio.run()** is used.

Example:

```python
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens)
    loop.close()
```

*New in version 3.6.*

**Scheduling callbacks**

loop. **call_soon(callback, *args, context=None)**
Schedule a **callback** to be called with **args** arguments at the next iteration of the event loop.

Callbacks are called in the order in which they are registered. Each callback will be called exactly once.

An optional keyword-only **context** argument allows specifying a custom **contextvars.Context** for the **callback** to run in. The
current context is used when no context is provided.

An instance of `asyncio.Handle` is returned, which can be used later to cancel the callback.

This method is not thread-safe.

```python
loop.call_soon_threadsafe(callback, *args, context=None)
```

A thread-safe variant of `call_soon()`. Must be used to schedule callbacks from another thread.

See the concurrency and multithreading section of the documentation.

*Changed in version 3.7:* The context keyword-only parameter was added. See [PEP 567](https://www.python.org/dev/peps/pep-0567/) for more details.

**Note:** Most `asyncio` scheduling functions don’t allow passing keyword arguments. To do that, use `functools.partial()`:

```python
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

Using partial objects is usually more convenient than using lambdas, as `asyncio` can render partial objects better in debug and error messages.

### Scheduling delayed callbacks

Event loop provides mechanisms to schedule callback functions to be called at some point in the future. Event loop uses monotonic clocks to
track time.

**loop.call_later**(*delay, callback, *args, context=*)

Schedule *callback* to be called after the given *delay* number of seconds (can be either an int or a float).

An instance of *asyncio.TimerHandle* is returned which can be used to cancel the callback.

*callback* will be called exactly once. If two callbacks are scheduled for exactly the same time, the order in which they are called is undefined.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with keyword arguments use *functools.partial()*.

An optional keyword-only *context* argument allows specifying a custom *contextvars.Context* for the *callback* to run in. The current context is used when no *context* is provided.

*Changed in version 3.7:* The *context* keyword-only parameter was added. See **PEP 567** for more details.

*Changed in version 3.7.1:* In Python 3.7.0 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.7.1.

**loop.call_at**(*when, callback, *args, context=*)

Schedule *callback* to be called at the given absolute timestamp *when* (an int or a float), using the same time reference as *loop.time()*.

This method's behavior is the same as **call_later**().
An instance of `asyncio.TimerHandle` is returned which can be used to cancel the callback.

*Changed in version 3.7:* The `context` keyword-only parameter was added. See [PEP 567](https://www.python.org/dev/peps/pep-0567/) for more details.

*Changed in version 3.7.1:* In Python 3.7.0 and earlier with the default event loop implementation, the difference between `when` and the current time could not exceed one day. This has been fixed in Python 3.7.1.

```python
loop.time()
```

Return the current time, as a `float` value, according to the event loop’s internal monotonic clock.

**Note:** Timeouts (relative `delay` or absolute `when`) should not exceed one day.

**See also:** The `asyncio.sleep()` function.

## Creating Futures and Tasks

```python
loop.create_future()
```

Create an `asyncio.Future` object attached to the event loop.

This is the preferred way to create Futures in asyncio. This lets third-party event loops provide alternative implementations of the Future object (with better performance or instrumentation).

*New in version 3.5.2.*

```python
loop.create_task(coro)
```

Schedule the execution of a Coroutines. Return a Task object.
Third-party event loops can use their own subclass of Task for interoperability. In this case, the result type is a subclass of Task.

```
loop.set_task_factory(factory)
```

Set a task factory that will be used by loop.create_task().

If `factory` is `None` the default task factory will be set. Otherwise, `factory` must be a callable with the signature matching `(loop, coro)`, where `loop` is a reference to the active event loop, and `coro` is a coroutine object. The callable must return a asyncio.Future-compatible object.

```
loop.get_task_factory()
```

Return a task factory or `None` if the default one is in use.

### Opening network connections

```
coroutine loop.create_connection(protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None)
```

Open a streaming transport connection to a given address specified by `host` and `port`.

The socket family can be either AF_INET or AF_INET6 depending on `host` (or the `family` argument, if provided).

The socket type will be SOCK_STREAM.

`protocol_factory` must be a callable returning an asyncio protocol implementation.

This method will try to establish the connection in the background.
When successful, it returns a \((\text{transport, protocol})\) pair.

The chronological synopsis of the underlying operation is as follows:

1. The connection is established and a \text{transport} is created for it.
2. \text{protocol\_factory} is called without arguments and is expected to return a \text{protocol} instance.
3. The protocol instance is coupled with the transport by calling its \text{connection\_made()} method.
4. A \((\text{transport, protocol})\) tuple is returned on success.

The created transport is an implementation-dependent bidirectional stream.

Other arguments:

- \text{ssl}: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If \text{ssl} is a \text{ssl\_SSLContext} object, this context is used to create the transport; if \text{ssl} is \text{True}, a default context returned from \text{ssl\_create\_default\_context()} is used.

**See also:** SSL/TLS security considerations

- \text{server\_hostname} sets or overrides the hostname that the target server’s certificate will be matched against. Should only be passed if \text{ssl} is not None. By default the value of the \text{host} argument is used. If \text{host} is empty, there is no default and you must pass a value for \text{server\_hostname}. If \text{server\_hostname} is an empty string, hostname matching is disabled (which is a serious security risk, allowing for potential man-in-the-middle attacks).
- `family`, `proto`, `flags` are the optional address family, protocol and flags to be passed through to `getaddrinfo()` for `host` resolution. If given, these should all be integers from the corresponding `socket` module constants.

- `sock`, if given, should be an existing, already connected `socket.socket` object to be used by the transport. If `sock` is given, none of `host`, `port`, `family`, `proto`, `flags` and `local_addr` should be specified.

- `local_addr`, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The `local_host` and `local_port` are looked up using `getaddrinfo()`, similarly to `host` and `port`.

- `ssl_handshake_timeout` is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. 60.0 seconds if `None` (default).

*New in version 3.7:* The `ssl_handshake_timeout` parameter.

*Changed in version 3.6:* The socket option `TCP_NODELAY` is set by default for all TCP connections.

*Changed in version 3.5:* Added support for SSL/TLS in `ProactorEventLoop`.

*See also:* The `open_connection()` function is a high-level alternative API. It returns a pair of `(StreamReader, StreamWriter)` that can be used directly in async/await code.

coroutine
loop. `create_datagram_endpoint(protocol_factory,`
local_addr=None, remote_addr=None, *, family=0, proto=0, flags=0, reuse_address=None, reuse_port=None, allow_broadcast=None, sock=None)

Create a datagram connection.

The socket family can be either AF_INET, AF_INET6, or AF_UNIX, depending on host (or the family argument, if provided).

The socket type will be SOCK_DGRAM.

protocol_factory must be a callable returning a protocol implementation.

A tuple of (transport, protocol) is returned on success.

Other arguments:

- local_addr, if given, is a (local_host, local_port) tuple used to bind the socket to locally. The local_host and local_port are looked up using getaddrinfo().
- remote_addr, if given, is a (remote_host, remote_port) tuple used to connect the socket to a remote address. The remote_host and remote_port are looked up using getaddrinfo().
- family, proto, flags are the optional address family, protocol and flags to be passed through to getaddrinfo() for host resolution. If given, these should all be integers from the corresponding socket module constants.
- reuse_address tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire. If not specified will automatically be set to True on Unix.
- reuse_port tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so
long as they all set this flag when being created. This option is 
not supported on Windows and some Unixes. If the 
SO_REUSEPORT constant is not defined then this capability is 
unsupported.

- allow_broadcast tells the kernel to allow this endpoint to send 
messages to the broadcast address.
- sock can optionally be specified in order to use a preexisting, 
already connected, socket.socket object to be used by the 
transport. If specified, local_addr and remote_addr should be 
 omitted (must be None).

On Windows, with ProactorEventLoop, this method is not 
supported.

See UDP echo client protocol and UDP echo server protocol 
examples.

*Changed in version 3.4.4:* The family, proto, flags, reuse_address, 
reuse_port, *allow_broadcast, and sock parameters were added.*

**coroutine loop.**

```
create_unix_connection(protocol_factory, 
path=None, *, ssl=None, sock=None, server_hostname=None, 
ssl_handshake_timeout=None)
```

Create a Unix connection.

The socket family will be AF_UNIX; socket type will be 
SOCK_STREAM.

A tuple of (transport, protocol) is returned on success.

*path* is the name of a Unix domain socket and is required, unless a 
*sock* parameter is specified. Abstract Unix sockets, str, bytes, 
and Path paths are supported.
See the documentation of the `loop.create_connection()` method for information about arguments to this method.

**Availability:** Unix.

*New in version 3.7:* The `ssl_handshake_timeout` parameter.

*Changed in version 3.7:* The `path` parameter can now be a path-like object.

### Creating network servers

```python
coroutine loop.create_server(protocol_factory, host=None, port=None, *, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, ssl_handshake_timeout=None, start_serving=True)
```

Create a TCP server (socket type `SOCK_STREAM`) listening on `port` of the `host` address.

Returns a `Server` object.

**Arguments:**

- `protocol_factory` must be a callable returning a `protocol` implementation.
- The `host` parameter can be set to several types which determine where the server would be listening:
  - If `host` is a string, the TCP server is bound to a single network interface specified by `host`.
  - If `host` is a sequence of strings, the TCP server is bound to all network interfaces specified by the sequence.
  - If `host` is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- `family` can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set, the `family` will be determined from host name (defaults to `AF_UNSPEC`).
- `flags` is a bitmask for `getaddrinfo()`.
- `sock` can optionally be specified in order to use a preexisting socket object. If specified, `host` and `port` must not be specified.
- `backlog` is the maximum number of queued connections passed to `listen()` (defaults to 100).
- `ssl` can be set to an `SSLContext` instance to enable TLS over the accepted connections.
- `reuse_address` tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on Unix.
- `reuse_port` tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.
- `ssl_handshake_timeout` is (for a TLS server) the time in seconds to wait for the TLS handshake to complete before aborting the connection. 60.0 seconds if `None` (default).
- `start_serving` set to `True` (the default) causes the created server to start accepting connections immediately. When set to `False`, the user should await on `Server.start_serving()` or `Server.serve_forever()` to make the server to start accepting connections.

*New in version 3.7:* Added `ssl_handshake_timeout` and `start_serving` parameters.

*Changed in version 3.6:* The socket option TCP_NODELAY is set by default for all TCP connections.
Changed in version 3.5: Added support for SSL/TLS in ProactorEventLoop.

Changed in version 3.5.1: The host parameter can be a sequence of strings.

See also: The start_server() function is a higher-level alternative API that returns a pair of StreamReader and StreamWriter that can be used in an async/await code.

coroutine loop.create_unix_server(protocol_factory, path=None, *, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, start_serving=True)

Similar to loop.create_server() but works with the AF_UNIX socket family.

path is the name of a Unix domain socket, and is required, unless a sock argument is provided. Abstract Unix sockets, str, bytes, and Path paths are supported.

See the documentation of the loop.create_server() method for information about arguments to this method.

Availability: Unix.

New in version 3.7: The ssl_handshake_timeout and start_serving parameters.

Changed in version 3.7: The path parameter can now be a Path object.

coroutine loop.connect_accepted_socket(protocol_factory, sock, *, ssl=None, ssl_handshake_timeout=None)
Wrap an already accepted connection into a transport/protocol pair.

This method can be used by servers that accept connections outside of asyncio but that use asyncio to handle them.

Parameters:

- `protocol_factory` must be a callable returning a protocol implementation.
- `sock` is a preexisting socket object returned from `socket.accept`.
- `ssl` can be set to an `SSLContext` to enable SSL over the accepted connections.
- `ssl_handshake_timeout` is (for an SSL connection) the time in seconds to wait for the SSL handshake to complete before aborting the connection. 60.0 seconds if `None` (default).

Returns a `(transport, protocol)` pair.

*New in version 3.7:* The `ssl_handshake_timeout` parameter.

*New in version 3.5.3.*

**Transferring files**

`coroutine loop . sendfile(transport, file, offset=0, count=None, *, fallback=True)`

Send a `file` over a `transport`. Return the total number of bytes sent.

The method uses high-performance `os.sendfile()` if available.

`file` must be a regular file object opened in binary mode.

`offset` tells from where to start reading the file. If specified, `count` is the total number of bytes to transmit as opposed to sending the file
until EOF is reached. File position is always updated, even when this method raises an error, and \texttt{file.tell()} can be used to obtain the actual number of bytes sent.

\emph{fallback} set to \texttt{True} makes asyncio to manually read and send the file when the platform does not support the sendfile system call (e.g. Windows or SSL socket on Unix).

Raise \texttt{SendfileNotAvailableError} if the system does not support the \emph{sendfile} syscall and \texttt{fallback} is \texttt{False}.

\emph{New in version 3.7.}

**TLS Upgrade**

\texttt{coroutine loop.start_tls(transport, protocol, sslcontext, *,
server_side=False, server_hostname=None,
ssl_handshake_timeout=None)}

Upgrade an existing transport-based connection to TLS.

Return a new transport instance, that the \texttt{protocol} must start using immediately after the \texttt{await}. The \texttt{transport} instance passed to the \texttt{start_tls} method should never be used again.

Parameters:

- \texttt{transport} and \texttt{protocol} instances that methods like \texttt{create_server()} and \texttt{create_connection()} return.
- \texttt{sslcontext}: a configured instance of \texttt{SSLContext}.
- \texttt{server_side} pass \texttt{True} when a server-side connection is being upgraded (like the one created by \texttt{create_server()}).
- \texttt{server_hostname}: sets or overrides the host name that the target server's certificate will be matched against.
SSL Handshake Timeout (ssl_handshake_timeout) is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. 60.0 seconds if None (default).

*New in version 3.7.*

**Watching file descriptors**

```python
loop.add_reader(fd, callback, *args)
```

Start monitoring the `fd` file descriptor for read availability and invoke `callback` with the specified arguments once `fd` is available for reading.

```python
loop.remove_reader(fd)
```

Stop monitoring the `fd` file descriptor for read availability.

```python
loop.add_writer(fd, callback, *args)
```

Start monitoring the `fd` file descriptor for write availability and invoke `callback` with the specified arguments once `fd` is available for writing.

Use `functools.partial()` to pass keyword arguments to `callback`.

```python
loop.remove_writer(fd)
```

Stop monitoring the `fd` file descriptor for write availability.

See also `Platform Support` section for some limitations of these methods.

**Working with socket objects directly**

In general, protocol implementations that use transport-based APIs
such as `loop.create_connection()` and `loop.create_server()` are faster than implementations that work with sockets directly. However, there are some use cases when performance is not critical, and working with `socket` objects directly is more convenient.

**coroutine loop.**

```
sock_recv(sock, nbytes)
```

Receive up to `nbytes` from `sock`. Asynchronous version of `socket.recv()`.

Return the received data as a bytes object.

- `sock` must be a non-blocking socket.

*Changed in version 3.7:* Even though this method was always documented as a coroutine method, releases before Python 3.7 returned a `Future`. Since Python 3.7 this is an `async def` method.

```
sock_recv_into(sock, buf)
```

Receive data from `sock` into the `buf` buffer. Modeled after the blocking `socket.recv_into()` method.

Return the number of bytes written to the buffer.

- `sock` must be a non-blocking socket.

*New in version 3.7.*

```
sock_sendall(sock, data)
```

Send `data` to the `sock` socket. Asynchronous version of `socket.sendall()`.

This method continues to send to the socket until either all data in
data has been sent or an error occurs. None is returned on success. On error, an exception is raised. Additionally, there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

sock must be a non-blocking socket.

Changed in version 3.7: Even though the method was always documented as a coroutine method, before Python 3.7 it returned an Future. Since Python 3.7, this is an async def method.

coroutine loop. **sock_connect**(sock, address)

Connect sock to a remote socket at address.

Asynchronous version of socket.connect().

sock must be a non-blocking socket.

Changed in version 3.5.2: address no longer needs to be resolved. sock_connect will try to check if the address is already resolved by calling socket.inet_pton(). If not, loop.getaddrinfo() will be used to resolve the address.

See also: loop.create_connection() and asyncio.open_connection().

coroutine loop. **sock_accept**(sock)

Accept a connection. Modeled after the blocking socket.accept() method.

The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the
connection, and address is the address bound to the socket on the other end of the connection.

sock must be a non-blocking socket.

*Changed in version 3.7:* Even though the method was always documented as a coroutine method, before Python 3.7 it returned a Future. Since Python 3.7, this is an async def method.*

See also: loop.create_server() and start_server().

coroutine loop.sock_sendfile(sock, file, offset=0, count=None, *, fallback=True)

Send a file using high-performance os.sendfile if possible. Return the total number of bytes sent.

Asynchronous version of socket.sendfile().

sock must be a non-blocking socket.SOCK_STREAM socket.

file must be a regular file object open in binary mode.

offset tells from where to start reading the file. If specified, count is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and file.tell() can be used to obtain the actual number of bytes sent.

fallback, when set to True, makes asyncio manually read and send the file when the platform does not support the sendfile syscall (e.g. Windows or SSL socket on Unix).

Raise SendfileNotAvailableError if the system does not
support `sendfile` syscall and `fallback` is `False`.

`sock` must be a non-blocking socket.

*New in version 3.7.*

**DNS**

`coroutine` loop. `getaddrinfo(host, port, *, family=0, type=0, proto=0, flags=0)`

Asynchronous version of `socket.getaddrinfo()`.

`coroutine` loop. `getnameinfo(sockaddr, flags=0)`

Asynchronous version of `socket.getnameinfo()`.

*Changed in version 3.7:* Both `getaddrinfo` and `getnameinfo` methods were always documented to return a coroutine, but prior to Python 3.7 they were, in fact, returning `asyncio.Future` objects. Starting with Python 3.7 both methods are coroutines.

**Working with pipes**

`coroutine` loop. `connect_read_pipe(protocol_factory, pipe)`

Register the read end of `pipe` in the event loop.

`protocol_factory` must be a callable returning an `asyncio` protocol implementation.

`pipe` is a file-like object.

Return pair `(transport, protocol)`, where `transport` supports the `ReadTransport` interface and `protocol` is an object instantiated by the `protocol_factory`. 
With **SelectorEventLoop** event loop, the *pipe* is set to non-blocking mode.

**coroutine loop**.\texttt{connect_write_pipe}(\texttt{protocol\_factory}, \texttt{pipe})

Register the write end of *pipe* in the event loop.

\textit{protocol\_factory} must be a callable returning an *asyncio protocol* implementation.

*pipe* is *file-like object*.

Return pair \((\texttt{transport}, \texttt{protocol})\), where \texttt{transport} supports \texttt{WriteTransport} interface and \texttt{protocol} is an object instantiated by the \texttt{protocol\_factory}.

With **SelectorEventLoop** event loop, the *pipe* is set to non-blocking mode.

**Note**: **SelectorEventLoop** does not support the above methods on Windows. Use **ProactorEventLoop** instead for Windows.

**See also**: The \texttt{loop.subprocess.exec()} and \texttt{loop.subprocess.shell()} methods.

**Unix signals**

**loop**.\texttt{add\_signal\_handler}(\texttt{signum}, \texttt{callback}, *\texttt{args}*)

Set \texttt{callback} as the handler for the \texttt{signum} signal.

The callback will be invoked by \texttt{loop}, along with other queued callbacks and runnable coroutines of that event loop. Unlike signal handlers registered using \texttt{signal.signal()}, a callback
registered with this function is allowed to interact with the event loop.

Raise `ValueError` if the signal number is invalid or uncatchable. Raise `RuntimeError` if there is a problem setting up the handler.

Use `functools.partial()` to pass keyword arguments to `callback`.

Like `signal.signal()`, this function must be invoked in the main thread.

`loop.remove_signal_handler(sig)`

Remove the handler for the `sig` signal.

Return `True` if the signal handler was removed, or `False` if no handler was set for the given signal.

**Availability**: Unix.

**See also**: The `signal` module.

**Executing code in thread or process pools**

`awaitable loop.run_in_executor(executor, func, *args)`

Arrange for `func` to be called in the specified executor.

The `executor` argument should be an `concurrent.futures.Executor` instance. The default executor is used if `executor` is `None`.

Example:
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor():
        result = await loop.run_in_executor(pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor():
        result = await loop.run_in_executor(pool, cpu_bound)
        print('custom process pool', result)
This method returns an `asyncio.Future` object.

Use `functools.partial()` to pass keyword arguments to `func`.

*Changed in version 3.5.3:* `loop.run_in_executor()` no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (`ThreadPoolExecutor`) to set the default.

`loop.set_default_executor(executor)`

Set `executor` as the default executor used by `run_in_executor()`. `executor` should be an instance of `ThreadPoolExecutor`.

*Deprecated since version 3.7:* Using an executor that is not an instance of `ThreadPoolExecutor` is deprecated and will trigger an error in Python 3.9.

`executor` must be an instance of `concurrent.futures.ThreadPoolExecutor`.

**Error Handling API**

Allows customizing how exceptions are handled in the event loop.

`loop.set_exception_handler(handler)`

Set `handler` as the new event loop exception handler.

If `handler` is `None`, the default exception handler will be set. Otherwise, `handler` must be a callable with the signature matching
(loop, context), where loop is a reference to the active event loop, and context is a dict object containing the details of the exception (see call_exception_handler() documentation for details about context).

`loop.get_exception_handler()`
Return the current exception handler, or None if no custom exception handler was set.

*New in version 3.5.2.*

`loop.default_exception_handler(context)`
Default exception handler.

This is called when an exception occurs and no exception handler is set. This can be called by a custom exception handler that wants to defer to the default handler behavior.

context parameter has the same meaning as in call_exception_handler().

`loop.call_exception_handler(context)`
Call the current event loop exception handler.

context is a dict object containing the following keys (new keys may be introduced in future Python versions):

- ‘message’: Error message;
- ‘exception’ (optional): Exception object;
- ‘future’ (optional): asyncio.Future instance;
- ‘handle’ (optional): asyncio.Handle instance;
- ‘protocol’ (optional): Protocol instance;
- ‘transport’ (optional): Transport instance;
• ‘socket’ (optional): `socket.socket` instance.

**Note:** This method should not be overloaded in subclassed event loops. For custom exception handling, use the `set_exception_handler()` method.

## Enabling debug mode

`loop.get_debug()`  
Get the debug mode (`bool`) of the event loop.

The default value is `True` if the environment variable `PYTHONASYNCIODEBUG` is set to a non-empty string, `False` otherwise.

`loop.set_debug(enabled: bool)`  
Set the debug mode of the event loop.

*Changed in version 3.7:* The new `-X dev` command line option can now also be used to enable the debug mode.

**See also:** The debug mode of asyncio.

## Running Subprocesses

Methods described in this subsections are low-level. In regular `async/await` code consider using the high-level `asyncio.create_subprocess_shell()` and `asyncio.create_subprocess_exec()` convenience functions instead.

**Note:** The default asyncio event loop on **Windows** does not
support subprocesses. See Subprocess Support on Windows for details.

coroutine loop. **subprocess_exec**(protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)

Create a subprocess from one or more string arguments specified by `args`.

`args` must be a list of strings represented by:

- `str`;
- or `bytes`, encoded to the filesystem encoding.

The first string specifies the program executable, and the remaining strings specify the arguments. Together, string arguments form the `argv` of the program.

This is similar to the standard library `subprocess.Popen` class called with `shell=False` and the list of strings passed as the first argument; however, where `Popen` takes a single argument which is list of strings, `subprocess_exec` takes multiple string arguments.

The `protocol_factory` must be a callable returning a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- `stdin`: either a file-like object representing a pipe to be connected to the subprocess's standard input stream using `connect_write_pipe()`, or the `subprocess.PIPE` constant (default). By default a new pipe will be created and connected.
- **stdout**: either a file-like object representing the pipe to be connected to the subprocess's standard output stream using `connect_read_pipe()`, or the `subprocess.PIPE` constant (default). By default a new pipe will be created and connected.

- **stderr**: either a file-like object representing the pipe to be connected to the subprocess's standard error stream using `connect_read_pipe()`, or one of `subprocess.PIPE` (default) or `subprocess.STDOUT` constants.

  By default a new pipe will be created and connected. When `subprocess.STDOUT` is specified, the subprocess' standard error stream will be connected to the same pipe as the standard output stream.

- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for `bufsize`, `universal_newlines` and `shell`, which should not be specified at all.

See the constructor of the `subprocess.Popen` class for documentation on other arguments.

Returns a pair of `(transport, protocol)`, where `transport` conforms to the `asyncio.SubprocessTransport` base class and `protocol` is an object instantiated by the `protocol_factory`.

**coroutine loop**. `subprocess_shell(protocol_factory, cmd, *,
stdin=subprocess.PIPE, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, **kwargs)`

Create a subprocess from `cmd`, which can be a `str` or a `bytes` string encoded to the `filesystem_encoding`, using the platform's
“shell” syntax.

This is similar to the standard library `subprocess.Popen` class called with `shell=True`.

The `protocol_factory` must be a callable returning a subclass of the `SubprocessProtocol` class.

See `subprocess_exec()` for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where `transport` conforms to the `SubprocessTransport` base class and `protocol` is an object instantiated by the `protocol_factory`.

**Note:** It is the application’s responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special characters in strings that are going to be used to construct shell commands.
Callback Handles

class asyncio.Handle
A callback wrapper object returned by loop.call_soon(),
loop.call_soon_threadsafe().

cancel()
Cancel the callback. If the callback has already been canceled
or executed, this method has no effect.

cancelled()
Return True if the callback was cancelled.

New in version 3.7.

class asyncio.TimerHandle
A callback wrapper object returned by loop.call_later(), and
loop.call_at().

This class is a subclass of Handle.

when()
Return a scheduled callback time as float seconds.

The time is an absolute timestamp, using the same time
reference as loop.time().

New in version 3.7.
Server Objects

Server objects are created by `loop.create_server()`, `loop.create_unix_server()`, `start_server()`, and `start_unix_server()` functions.

Do not instantiate the class directly.

```python
class asyncio.Server

Server objects are asynchronous context managers. When used in an `async with` statement, it’s guaranteed that the Server object is closed and not accepting new connections when the `async with` statement is completed:

```python
srv = await loop.create_server(...)  

async with srv:
    # some code

    # At this point, srv is closed and no longer accepting new connections.
```

*Changed in version 3.7:* Server object is an asynchronous context manager since Python 3.7.

`close()`

Stop serving: close listening sockets and set the `sockets` attribute to `None`.

The sockets that represent existing incoming client connections are left open.

The server is closed asynchronously, use the `wait_closed()`
coroutine to wait until the server is closed.

**get_loop()**
Return the event loop associated with the server object.

*New in version 3.7.*

**coroutine start_serving()**
Start accepting connections.

This method is idempotent, so it can be called when the server is already being serving.

The *start_serving* keyword-only parameter to *loop.create_server()* and *asyncio.start_server()* allows creating a Server object that is not accepting connections initially. In this case *Server.start_serving()*, or *Server.serve_forever()* can be used to make the Server start accepting connections.

*New in version 3.7.*

**coroutine serve_forever()**
Start accepting connections until the coroutine is cancelled. Cancellation of *serve_forever* task causes the server to be closed.

This method can be called if the server is already accepting connections. Only one *serve_forever* task can exist per one *Server* object.

Example:

```python
async def client_connected(reader, writer):
```
# Communicate with the client with reader/writer streams. For example:

```python
async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

New in version 3.7.

**is_serving()**

Return `True` if the server is accepting new connections.

New in version 3.7.

Coroutine **wait_closed()**

Wait until the `close()` method completes.

**sockets**

List of `socket.socket` objects the server is listening on, or `None` if the server is closed.

Changed in version 3.7: Prior to Python 3.7 `Server.sockets` used to return an internal list of server sockets directly. In 3.7 a copy of that list is returned.
Event Loop Implementations

asyncio ships with two different event loop implementations: SelectorEventLoop and ProactorEventLoop.

By default asyncio is configured to use SelectorEventLoop on all platforms.

class asyncio.**SelectorEventLoop**

An event loop based on the selectors module.

Uses the most efficient selector available for the given platform. It is also possible to manually configure the exact selector implementation to be used:

```python
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

**Availability:** Unix, Windows.

class asyncio.**ProactorEventLoop**

An event loop for Windows that uses “I/O Completion Ports” (IOCP).

**Availability:** Windows.

An example how to use ProactorEventLoop on Windows:

```python
import asyncio
```
import sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)

See also: MSDN documentation on I/O Completion Ports.

class asyncio.AbstractEventLoop
    Abstract base class for asyncio-compliant event loops.

    The Event Loop Methods section lists all methods that an alternative implementation of AbstractEventLoop should have defined.
Examples

Note that all examples in this section purposefully show how to use the low-level event loop APIs, such as `loop.run_forever()` and `loop.call_soon()`. Modern asyncio applications rarely need to be written this way; consider using the high-level functions like `asyncio.run()`.

Hello World with call_soon()

An example using the `loop.call_soon()` method to schedule a callback. The callback displays "Hello World" and then stops the event loop:

```python
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```
Display the current date with call_later()

An example of a callback displaying the current date every second. The callback uses the `loop.call_later()` method to reschedule itself after 5 seconds, and then stops the event loop:

```python
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

See also: A similar current date example created with a coroutine and the `run()` function.
Watch a file descriptor for read events

Wait until a file descriptor received some data using the `loop.add_reader()` method and then close the event loop:

```python
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received: ", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
```
See also:

- A similar example using transports, protocols, and the `loop.create_connection()` method.
- Another similar example using the high-level `asyncio.open_connection()` function and streams.

Set signal handlers for SIGINT and SIGTERM

(This `signals` example only works on Unix.)

Register handlers for signals SIGINT and SIGTERM using the `loop.add_signal_handler()` method:

```python
import asyncio
import functools
import os
import signal

def ask_exit(signame):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in ['SIGINT', 'SIGTERM']:
        loop.add_signal_handler(
            signal.getattr(signal, signame),
            functools.partial(ask_exit, signame))

    await asyncio.sleep(3600)
```
print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")
asyncio.run(main())
Futures

*Future* objects are used to bridge low-level callback-based code with high-level async/await code.
Future Functions

asyncio.isfuture(obj)

Return True if obj is either of:

- an instance of asyncio.Future,
- an instance of asyncio.Task,
- a Future-like object with a _asyncio_future_blocking attribute.

*New in version 3.5.*

asyncio.ensure_future(obj, *, loop=None)

Return:

- obj argument as is, if obj is a Future, a Task, or a Future-like object (isfuture() is used for the test.)
- a Task object wrapping obj, if obj is a coroutine (iscoroutine() is used for the test.)
- a Task object that would await on obj, if obj is an awaitable (inspect.isawaitable() is used for the test.)

If obj is neither of the above a TypeError is raised.

**Important:** See also the create_task() function which is the preferred way for creating new Tasks.

*Changed in version 3.5.1:* The function accepts any awaitable object.

asyncio.wrap_future(future, *, loop=None)

Future Object

class asyncio.Future(*, loop=None)

A Future represents an eventual result of an asynchronous operation. Not thread-safe.

Future is an awaitable object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled.

Typically Futures are used to enable low-level callback-based code (e.g. in protocols implemented using asyncio transports) to interoperate with high-level async/await code.

The rule of thumb is to never expose Future objects in user-facing APIs, and the recommended way to create a Future object is to call loop.create_future(). This way alternative event loop implementations can inject their own optimized implementations of a Future object.

Changed in version 3.7: Added support for the contextvars module.

result()

Return the result of the Future.

If the Future is done and has a result set by the set_result() method, the result value is returned.

If the Future is done and has an exception set by the set_exception() method, this method raises the exception.

If the Future has been cancelled, this method raises a
CancelledError exception.

If the Future's result isn't yet available, this method raises a InvalidStateError exception.

**set_result(result)**
Mark the Future as done and set its result.

Raises a InvalidStateError error if the Future is already done.

**set_exception(exception)**
Mark the Future as done and set an exception.

Raises a InvalidStateError error if the Future is already done.

**done()**
Return True if the Future is done.

A Future is done if it was cancelled or if it has a result or an exception set with set_result() or set_exception() calls.

**cancelled()**
Return True if the Future was cancelled.

The method is usually used to check if a Future is not cancelled before setting a result or an exception for it:

```python
if not fut.cancelled():
    fut.set_result(42)
```

**add_done_callback(callback, *, context=None)**
Add a callback to be run when the Future is done.

The callback is called with the Future object as its only argument.

If the Future is already done when this method is called, the callback is scheduled with loop.call_soon().

An optional keyword-only context argument allows specifying a custom contextvars.Context for the callback to run in. The current context is used when no context is provided.

functools.partial() can be used to pass parameters to the callback, e.g.:

```python
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Changed in version 3.7: The context keyword-only parameter was added. See PEP 567 for more details.

remove_done_callback(callback)
Remove callback from the callbacks list.

Returns the number of callbacks removed, which is typically 1, unless a callback was added more than once.

cancel()
Cancel the Future and schedule callbacks.

If the Future is already done or cancelled, return False. Otherwise, change the Future's state to cancelled, schedule the
callbacks, and return `True`.

**exception()**

Return the exception that was set on this Future.

The exception (or `None` if no exception was set) is returned only if the Future is *done*.

If the Future has been *cancelled*, this method raises a `CancelledError` exception.

If the Future isn’t *done* yet, this method raises an `InvalidStateError` exception.

**get_loop()**

Return the event loop the Future object is bound to.

*New in version 3.7.*

This example creates a Future object, creates and schedules an asynchronous Task to set result for the Future, and waits until the Future has a result:

```python
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
```
fut = loop.create_future()

# Run "set_after()" coroutine in a parallel Task.
# We are using the low-level "loop.create_task()"
# we already have a reference to the event loop.
# Otherwise we could have just used "asyncio.create_task().
loop.create_task(
    set_after(fut, 1, '... world'))

print('hello ...')

# Wait until *fut* has a result (1 second) and print it.
print(await fut)

asyncio.run(main())

**Important**: The Future object was designed to mimic `concurrent.futures.Future`. Key differences include:

- unlike asyncio Futures, `concurrent.futures.Future` instances cannot be awaited.
- `asyncio.Future.result()` and `asyncio.Future.exception()` do not accept the `timeout` argument.
- `asyncio.Future.result()` and `asyncio.Future.exception()` raise an `InvalidStateError` exception when the Future is not done.
- Callbacks registered with `asyncio.Future.add_done_callback()` are not called immediately. They are scheduled with `loop.call_soon()` instead.
- `asyncio.Future` is not compatible with the `concurrent.futures.wait()` and `concurrent.futures.as_completed()` functions.
Transports and Protocols

Preface

Transports and Protocols are used by the low-level event loop APIs such as loop.create_connection(). They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both Transports and Protocols.

Introduction

At the highest level, the transport is concerned with how bytes are transmitted, while the protocol determines which bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport’s point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.
Most of connection oriented event loop methods (such as `loop.create_connection()`) usually accept a `protocol_factory` argument used to create a `Protocol` object for an accepted connection, represented by a `Transport` object. Such methods usually return a tuple of `(transport, protocol).

Contents

This documentation page contains the following sections:


- The **Examples** section showcases how to work with transports, protocols, and low-level event loop APIs.
Transports

Transports are classes provided by asyncio in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an asyncio event loop. asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport’s kind.

The transport classes are not thread safe.

Transports Hierarchy

class asyncio.BaseTransport
Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.WriteTransport(BaseTransport)
A base transport for write-only connections.

Instances of the WriteTransport class are returned from the loop.connect_write_pipe() event loop method and are also used by subprocess-related methods like loop.subprocess_exec().

class asyncio.ReadTransport(BaseTransport)
A base transport for read-only connections.

Instances of the ReadTransport class are returned from the
loop.connect_read_pipe() event loop method and are also used by subprocess-related methods like loop.subprocess_exec().

class asyncio.Transport(WriteTransport, ReadTransport)
Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the Transport class are returned from or used by event loop methods like loop.create_connection(), loop.create_unix_connection(), loop.create_server(), loop.sendfile(), etc.

class asyncio.DatagramTransport(BaseTransport)
A transport for datagram (UDP) connections.

Instances of the DatagramTransport class are returned from the loop.create_datagram_endpoint() event loop method.

class asyncio.SubprocessTransport(BaseTransport)
An abstraction to represent a connection between a parent and its child OS process.

Instances of the SubprocessTransport class are returned from event loop methods loop.subprocess_shell() and loop.subprocess_exec().

Base Transport
**BaseTransport.** close()

Close the transport.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `protocol.connection_lost()` method will be called with `None` as its argument.

**BaseTransport.** is_closing()

Return `True` if the transport is closing or is closed.

**BaseTransport.** get_extra_info(name, default=None)

Return information about the transport or underlying resources it uses.

`name` is a string representing the piece of transport-specific information to get.

`default` is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```python
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
  - `peername`: the remote address to which the socket is
connected, result of `socket.socket.getpeername()` (None on error)
- 'socket': `socket.socket` instance
- 'sockname': the socket's own address, result of `socket.socket.getsockname()`

- SSL socket:
  - 'compression': the compression algorithm being used as a string, or None if the connection isn't compressed; result of `ssl.SSLSocket.compression()`
  - 'cipher': a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of `ssl.SSLSocket.cipher()`
  - 'peercert': peer certificate; result of `ssl.SSLSocket.getpeercert()`
  - 'sslcontext': `ssl.SSLContext` instance
  - 'ssl_object': `ssl.SSLObject` or `ssl.SSLSocket` instance

- pipe:
  - 'pipe': pipe object

- subprocess:
  - 'subprocess': `subprocess.Popen` instance

BaseTransport.set_protocol(protocol)
Set a new protocol.

Switching protocol should only be done when both protocols are documented to support the switch.

BaseTransport.get_protocol()
Return the current protocol.
Read-only Transports

ReadTransport.**is_reading()**
Return `True` if the transport is receiving new data.

*New in version 3.7.*

ReadTransport.**pause_reading()**
Pause the receiving end of the transport. No data will be passed to the protocol's `protocol.data_received()` method until `resume_reading()` is called.

*Changed in version 3.7:* The method is idempotent, i.e. it can be called when the transport is already paused or closed.

ReadTransport.**resume_reading()**
Resume the receiving end. The protocol's `protocol.data_received()` method will be called once again if some data is available for reading.

*Changed in version 3.7:* The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

WriteTransport.**abort()**
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `protocol.connection_lost()` method will eventually be called with `None` as its argument.

WriteTransport.**can_write_eof()**
Return `True` if the transport supports `write_eof()`, `False` if not.

**WriteTransport.get_write_buffer_size()**

Return the current size of the output buffer used by the transport.

**WriteTransport.get_write_buffer_limits()**

Get the `high` and `low` watermarks for write flow control. Return a tuple `(low, high)` where `low` and `high` are positive number of bytes.

Use `set_write_buffer_limits()` to set the limits.

*New in version 3.4.2.*

**WriteTransport.set_write_buffer_limits**(high=None, low=None)

Set the `high` and `low` watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's `protocol.pause_writing()` and `protocol.resume_writing()` methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither `high` nor `low` can be negative.

`pause_writing()` is called when the buffer size becomes greater than or equal to the `high` value. If writing has been paused, `resume_writing()` is called when the buffer size becomes less than or equal to the `low` value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high
watermark. Setting high to zero forces low to zero as well, and causes `pause_writing()` to be called whenever the buffer becomes non-empty. Setting low to zero causes `resume_writing()` to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use `get_write_buffer_limits()` to get the limits.

**WriteTransport**. `write(data)`

Write some `data` bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

**WriteTransport**. `writelines(list_of_data)`

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling `write()` on each element yielded by the iterable, but may be implemented more efficiently.

**WriteTransport**. `write_eof()`

Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise `NotImplementedError` if the transport (e.g. SSL) doesn't support half-closed connections.

**Datagram Transports**

**DatagramTransport**. `sendto(data, addr=None)`

Send the `data` bytes to the remote peer given by `addr` (a transport-dependent target address). If `addr` is `None`, the data is sent to the
target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

`DatagramTransport.abort()`  
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `protocol.connection_lost()` method will eventually be called with `None` as its argument.

**Subprocess Transports**

`SubprocessTransport.get_pid()`  
Return the subprocess process id as an integer.

`SubprocessTransport.get_pipe_transport(fd)`  
Return the transport for the communication pipe corresponding to the integer file descriptor `fd`:

- **0**: readable streaming transport of the standard input (`stdin`), or `None` if the subprocess was not created with `stdin=PIPE`  
- **1**: writable streaming transport of the standard output (`stdout`), or `None` if the subprocess was not created with `stdout=PIPE`  
- **2**: writable streaming transport of the standard error (`stderr`), or `None` if the subprocess was not created with `stderr=PIPE`  
- **other fd**: `None`

`SubprocessTransport.get_returncode()`  
Return the subprocess return code as an integer or `None` if it
hasn’t returned, which is similar to the `subprocess.Popen.returncode` attribute.

SubprocessTransport.**kill()**

Kill the subprocess.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for `terminate()`.

See also `subprocess.Popen.kill()`.

SubprocessTransport.**send_signal**(signal)

Send the `signal` number to the subprocess, as in `subprocess.Popen.send_signal()`.

SubprocessTransport.**terminate()**

Stop the subprocess.

On POSIX systems, this method sends SIGTERM to the subprocess. On Windows, the Windows API function `TerminateProcess()` is called to stop the subprocess.

See also `subprocess.Popen.terminate()`.

SubprocessTransport.**close()**

Kill the subprocess by calling the `kill()` method.

If the subprocess hasn’t returned yet, and close transports of `stdin`, `stdout`, and `stderr` pipes.
Protocols

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with transports.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Base Protocols

class asyncio.BaseProtocol
   Base protocol with methods that all protocols share.

class asyncio.Protocol(BaseProtocol)
   The base class for implementing streaming protocols (TCP, Unix sockets, etc).

class asyncio.BufferedProtocol(BaseProtocol)
   A base class for implementing streaming protocols with manual control of the receive buffer.

class asyncio.DatagramProtocol(BaseProtocol)
   The base class for implementing datagram (UDP) protocols.

class asyncio.SubprocessProtocol(BaseProtocol)
   The base class for implementing protocols communicating with child processes (unidirectional pipes).
Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

**Connection Callbacks**

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

**BaseProtocol**: `connection_made(transport)`

Called when a connection is made.

The `transport` argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

**BaseProtocol**: `connection_lost(exc)`

Called when the connection is lost or closed.

The argument is either an exception object or `None`. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

**Flow Control Callbacks**

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the `set_write_buffer_limits()` method for more details.

**BaseProtocol**: `pause_writing()`
Called when the transport's buffer goes over the high watermark.

BaseProtocol.

\texttt{resume\_writing()}

Called when the transport's buffer drains below the low watermark.

If the buffer size equals the high watermark, \texttt{pause\_writing()} is not called: the buffer size must go strictly over.

Conversely, \texttt{resume\_writing()} is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

\textbf{Streaming Protocols}

Event methods, such as \texttt{loop.create\_server()}, \texttt{loop.create\_unix\_server()}, \texttt{loop.create\_connection()}, \texttt{loop.create\_unix\_connection()}, \texttt{loop.connect\_accepted\_socket()}, \texttt{loop.connect\_read\_pipe()}, \texttt{loop.connect\_write\_pipe()} accept factories that return streaming protocols.

Protocol.

\texttt{data\_received(data)}

Called when some data is received. \texttt{data} is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.
The method can be called an arbitrary number of times while a connection is open.

However, `protocol.eof_received()` is called at most once. Once `eof_received()` is called, `data_received()` is not called anymore.

**Protocol.eof_received()**
Called when the other end signals it won’t send any more data (for example by calling `transport.write_eof()`, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don’t support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start  ->  connection_made
       [-> data_received] *
       [-> eof_received]?
->  connection_lost  ->  end
```

**Buffered Streaming Protocols**

*New in version 3.7: Important:* this has been added to asyncio in Python 3.7 on a provisional basis! This is as an experimental API that
might be changed or removed completely in Python 3.8.

Buffered Protocols can be used with any event loop method that supports Streaming Protocols.

BufferedProtocol implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on BufferedProtocol instances:

BufferedProtocol.get_buffer(sizehint)
   Called to allocate a new receive buffer.

   sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what sizehint suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

   get_buffer() must return an object implementing the buffer protocol.

BufferedProtocol.buffer_updated(nbytes)
   Called when the buffer was updated with the received data.

   nbytes is the total number of bytes that were written to the buffer.

BufferedProtocol.eof_received()
   See the documentation of the protocol.eof_received() method.
get_buffer() can be called an arbitrary number of times during a connection. However, protocol.eof_received() is called at most once and, if called, get_buffer() and buffer_updated() won't be called after it.

State machine:

```
start -> connection_made
   [-> get_buffer
      [-> buffer_updated]?]*
   [-> eof_received]?
-> connection_lost -> end
```

Datagram Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the loop.create_datagram_endpoint() method.

DatagramProtocol.datagram_received(data, addr)

Called when a datagram is received. data is a bytes object containing the incoming data. addr is the address of the peer sending the data; the exact format depends on the transport.

DatagramProtocol.error_received(exc)

Called when a previous send or receive operation raises an OSError. exc is the OSError instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.
Note: On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears ‘ready’ and excess packets are dropped. An OSError with errno set to errno.ENOBUFS may or may not be raised; if it is raised, it will be reported to DatagramProtocol.error_received() but otherwise ignored.

Subprocess Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the loop.subprocess_exec() and loop.subprocess_shell() methods.

SubprocessProtocol.pipe_data_received(fd, data)
Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

SubprocessProtocol.pipe_connection_lost(fd, exc)
Called when one of the pipes communicating with the child process is closed.

fd is the integer file descriptor that was closed.

SubprocessProtocol.process_exited()
Called when the child process has exited.
Examples

TCP Echo Server

Create a TCP echo server using the `loop.create_server()` method, send back received data, and close the connection:

```python
import asyncio

class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))
        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

async def main():
    # Get a reference to the event loop as we plan to
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)
```
async with server:
    await server.serve_forever()

tcpio.run(main())

See also: The TCP echo server using streams example uses the high-level asyncio.start_server() function.

TCP Echo Client

A TCP echo client using the loop.create_connection() method, sends data, and waits until the connection is closed:

```python
import asyncio
class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost, loop):
        self.message = message
        self.loop = loop
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)
```
async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()

asyncio.run(main())

See also: The TCP echo client using streams example uses the
high-level asyncio.open_connection() function.

UDP Echo Server

A UDP echo server, using the
loop.create_datagram_endpoint() method, sends back
received data:

import asyncio
class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour
    finally:
        transport.close()

asyncio.run(main())

UDP Echo Client
A UDP echo client, using the `loop.create_datagram_endpoint()` method, sends data and closes the transport when it receives the answer:

```python
import asyncio

class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None
        self.on_con_lost = loop.create_future()

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())
        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)

async def main():
    # Get a reference to the event loop as we plan to
    # low-level APIs.
```
```
loop = asyncio.get_running_loop()

message = "Hello World!"
transport, protocol = await loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=('127.0.0.1', 9999))

try:
    await protocol.on_con_lost
finally:
    transport.close()

asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the `loop.create_connection()` method with a protocol:

```
import asyncio
import socket

class MyProtocol(asyncio.Protocol):

    def __init__(self, loop):
        self.transport = None
        self.on_con_lost = loop.create_future()

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())
```
# We are done: close the transport;
# connection_lost() will be called automatically.
self.transport.close()

def connection_lost(self, exc):
    # The socket has been closed
    self.on_con_lost.set_result(True)

async def main():
    # Get a reference to the event loop as we plan to
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(loop), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())

See also: The watch a file descriptor for read events example uses the low-level loop.add_reader() method to register an FD.
The register an open socket to wait for data using streams example uses high-level streams created by the open_connection() function in a coroutine.

**loop.subprocess_exec() and SubprocessProtocol**

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by the `loop.subprocess_exec()` method:

```python
class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol
```

```python
```
# redirect the standard output into a pipe.
transport, protocol = await loop.subprocess_exec(
    lambda: DateProtocol(exit_future),
    sys.executable, '-c', code,
    stdin=None, stderr=None)

# Wait for the subprocess exit using the process
# method of the protocol.
await exit_future

# Close the stdout pipe.
transport.close()

# Read the output which was collected by the
# pipe_data_received() method of the protocol.
data = bytes(protocol.output)
return data.decode('ascii').rstrip()

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")

See also the same example written using high-level APIs.
Policies

An event loop policy is a global per-process object that controls the management of the event loop. Each event loop has a default policy, which can be changed and customized using the policy API.

A policy defines the notion of context and manages a separate event loop per context. The default policy defines context to be the current thread.

By using a custom event loop policy, the behavior of get_event_loop(), set_event_loop(), and new_event_loop() functions can be customized.

Policy objects should implement the APIs defined in the AbstractEventLoopPolicy abstract base class.
Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

```python
asyncio.get_event_loop_policy()
    Return the current process-wide policy.

asyncio.set_event_loop_policy(policy)
    Set the current process-wide policy to policy.

    If policy is set to None, the default policy is restored.
```
Policy Objects

The abstract event loop policy base class is defined as follows:

```python
class asyncio.AbstractEventLoopPolicy
    An abstract base class for asyncio policies.
```

- **get_event_loop()**
  - Get the event loop for the current context.
  - Return an event loop object implementing the `AbstractEventLoop` interface.
  - This method should never return `None`.
  - *Changed in version 3.6.*

- **set_event_loop(loop)**
  - Set the event loop for the current context to `loop`.

- **new_event_loop()**
  - Create and return a new event loop object.
  - This method should never return `None`.

- **get_child_watcher()**
  - Get a child process watcher object.
  - Return a watcher object implementing the `AbstractChildWatcher` interface.
  - This function is Unix specific.
set_child_watcher(watcher)

Set the current child process watcher to watcher.

This function is Unix specific.

asyncio ships with the following built-in policies:

class asyncio.DefaultEventLoopPolicy

The default asyncio policy. Uses SelectorEventLoop on both Unix and Windows platforms.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

class asyncio.WindowsProactorEventLoopPolicy

An alternative event loop policy that uses the ProactorEventLoop event loop implementation.

Availability: Windows.
Process Watchers

A process watcher allows customization of how an event loop monitors child processes on Unix. Specifically, the event loop needs to know when a child process has exited.

In asyncio, child processes are created with `create_subprocess_exec()` and `loop.subprocess_exec()` functions.

asyncio defines the `AbstractChildWatcher` abstract base class, which child watchers should implement, and has two different implementations: `SafeChildWatcher` (configured to be used by default) and `FastChildWatcher`.

See also the Subprocess and Threads section.

The following two functions can be used to customize the child process watcher implementation used by the asyncio event loop:

```python
asyncio.get_child_watcher()
```

Return the current child watcher for the current policy.

```python
asyncio.set_child_watcher(watcher)
```

Set the current child watcher to `watcher` for the current policy. `watcher` must implement methods defined in the `AbstractChildWatcher` base class.

**Note:** Third-party event loops implementations might not support custom child watchers. For such event loops, using `set_child_watcher()` might be prohibited or have no effect.
class asyncio.AbstractChildWatcher

add_child_handler(pid, callback, *args)

Register a new child handler.

Arrange for callback(pid, returncode, *args) to be called when a process with PID equal to pid terminates. Specifying another callback for the same process replaces the previous handler.

The callback callable must be thread-safe.

remove_child_handler(pid)

Removes the handler for process with PID equal to pid.

The function returns True if the handler was successfully removed, False if there was nothing to remove.

attach_loop(loop)

Attach the watcher to an event loop.

If the watcher was previously attached to an event loop, then it is first detached before attaching to the new loop.

Note: loop may be None.

close()

Close the watcher.

This method has to be called to ensure that underlying resources are cleaned-up.

class asyncio.SafeChildWatcher

This implementation avoids disrupting other code spawning processes by polling every process explicitly on a SIGCHLD signal.
This is a safe solution but it has a significant overhead when handling a big number of processes ($O(n)$ each time a SIGCHLD is received).

asyncio uses this safe implementation by default.

class asyncio.FastChildWatcher
This implementation reaps every terminated processes by calling os.waitpid(-1) directly, possibly breaking other code spawning processes and waiting for their termination.

There is no noticeable overhead when handling a big number of children ($O(1)$ each time a child terminates).
Custom Policies

To implement a new event loop policy, it is recommended to subclass DefaultEventLoopPolicy and override the methods for which custom behavior is wanted, e.g.:

```python
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):
    def get_event_loop(self):
        """Get the event loop.
        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```
Platform Support

The asyncio module is designed to be portable, but some platforms have subtle differences and limitations due to the platforms' underlying architecture and capabilities.
All Platforms

- `loop.add_reader()` and `loop.add_writer()` cannot be used to monitor file I/O.
Windows

All event loops on Windows do not support the following methods:

- `loop.create_unix_connection()` and `loop.create_unix_server()` are not supported. The `socket.AF_UNIX` socket family is specific to Unix.
- `loop.add_signal_handler()` and `loop.remove_signal_handler()` are not supported.

**SelectorEventLoop** has the following limitations:

- **SelectSelector** is used to wait on socket events: it supports sockets and is limited to 512 sockets.
- `loop.add_reader()` and `loop.add_writer()` only accept socket handles (e.g. pipe file descriptors are not supported).
- Pipes are not supported, so the `loop.connect_read_pipe()` and `loop.connect_write_pipe()` methods are not implemented.
- **Subprocesses** are not supported, i.e. `loop.subprocess_exec()` and `loop.subprocess_shell()` methods are not implemented.

**ProactorEventLoop** has the following limitations:

- The `loop.create_datagram_endpoint()` method is not supported.
- The `loop.add_reader()` and `loop.add_writer()` methods are not supported.

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows
Subprocess Support on Windows

SelectorEventLoop on Windows does not support subprocesses. On Windows, ProactorEventLoop should be used instead:

```python
import asyncio
asyncio.set_event_loop_policy(
    asyncio.WindowsProactorEventLoopPolicy())
asyncio.run(your_code())
```

The `policy.set_child_watcher()` function is also not supported, as ProactorEventLoop has a different mechanism to watch child processes.
macOS

Modern macOS versions are fully supported.

**macOS <= 10.8**

On macOS 10.6, 10.7 and 10.8, the default event loop uses `selectors.KqueueSelector`, which does not support character devices on these versions. The `SelectorEventLoop` can be manually configured to use `SelectSelector` or `PollSelector` to support character devices on these older versions of macOS. Example:

```python
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```
High-level API Index

This page lists all high-level async/await enabled asyncio APIs.
# Tasks

Utilities to run asyncio programs, create Tasks, and await on multiple things with timeouts.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run()</code></td>
<td>Create event loop, run a coroutine, close the loop.</td>
</tr>
<tr>
<td><code>create_task()</code></td>
<td>Start an asyncio Task.</td>
</tr>
<tr>
<td><code>await sleep()</code></td>
<td>Sleep for a number of seconds.</td>
</tr>
<tr>
<td><code>await gather()</code></td>
<td>Schedule and wait for things concurrently.</td>
</tr>
<tr>
<td><code>await wait_for()</code></td>
<td>Run with a timeout.</td>
</tr>
<tr>
<td><code>await shield()</code></td>
<td>Shield from cancellation.</td>
</tr>
<tr>
<td><code>await wait()</code></td>
<td>Monitor for completion.</td>
</tr>
<tr>
<td><code>current_task()</code></td>
<td>Return the current Task.</td>
</tr>
<tr>
<td><code>all_tasks()</code></td>
<td>Return all tasks for an event loop.</td>
</tr>
<tr>
<td>Task</td>
<td>Task object.</td>
</tr>
<tr>
<td><code>run_coroutine_threadsafe()</code></td>
<td>Schedule a coroutine from another OS thread.</td>
</tr>
<tr>
<td><code>for in as_completed()</code></td>
<td>Monitor for completion with a for loop.</td>
</tr>
</tbody>
</table>

## Examples

- Using asyncio.gather() to run things in parallel.
- Using asyncio.wait_for() to enforce a timeout.
- Cancellation.
- Using asyncio.sleep().
- See also the main Tasks documentation page.
Queues

Queues should be used to distribute work amongst multiple asyncio Tasks, implement connection pools, and pub/sub patterns.

<table>
<thead>
<tr>
<th>Queue</th>
<th>A FIFO queue.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue</td>
<td>A priority queue.</td>
</tr>
<tr>
<td>LifoQueue</td>
<td>A LIFO queue.</td>
</tr>
</tbody>
</table>

Examples

- Using asyncio.Queue to distribute workload between several Tasks.
- See also the Queues documentation page.
Subprocesses

Utilities to spawn subprocesses and run shell commands.

| await create_subprocess_exec() | Create a subprocess. |
| await create_subprocess_shell() | Run a shell command. |

Examples

- Executing a shell command.
- See also the subprocess APIs documentation.
 Streams

High-level APIs to work with network IO.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await open_connection()</code></td>
<td>Establish a TCP connection.</td>
</tr>
<tr>
<td><code>await open_unix_connection()</code></td>
<td>Establish a Unix socket connection.</td>
</tr>
<tr>
<td><code>await start_server()</code></td>
<td>Start a TCP server.</td>
</tr>
<tr>
<td><code>await start_unix_server()</code></td>
<td>Start a Unix socket server.</td>
</tr>
<tr>
<td>StreamReader</td>
<td>High-level async/await object to receive network data.</td>
</tr>
<tr>
<td>StreamWriter</td>
<td>High-level async/await object to send network data.</td>
</tr>
</tbody>
</table>

Examples

- Example TCP client.
- See also the streams APIs documentation.
Synchronization

Threading-like synchronization primitives that can be used in Tasks.

<table>
<thead>
<tr>
<th>Lock</th>
<th>A mutex lock.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>An event object.</td>
</tr>
<tr>
<td>Condition</td>
<td>A condition object.</td>
</tr>
<tr>
<td>Semaphore</td>
<td>A semaphore.</td>
</tr>
<tr>
<td>BoundedSemaphore</td>
<td>A bounded semaphore.</td>
</tr>
</tbody>
</table>

Examples

- Using asyncio.Event.
- See also the documentation of asyncio synchronization primitives.
## Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asyncio.TimeoutError</code></td>
<td>Raised on timeout by functions like <code>wait_for()</code>. Keep in mind that <code>asyncio.TimeoutError</code> is <strong>unrelated</strong> to the built-in <code>TimeoutError</code> exception.</td>
</tr>
<tr>
<td><code>asyncio.CancelledError</code></td>
<td>Raised when a Task is cancelled. See also <code>Task.cancel()</code>.</td>
</tr>
</tbody>
</table>

### Examples

- Handling `CancelledError` to run code on cancellation request.
- See also the full list of **asyncio-specific exceptions**.
Low-level API Index

This page lists all low-level asyncio APIs.
Obtaining the Event Loop

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asyncio.get_running_loop()</td>
<td>The preferred function to get the running event loop.</td>
</tr>
<tr>
<td>asyncio.get_event_loop()</td>
<td>Get an event loop instance (current or via the policy).</td>
</tr>
<tr>
<td>asyncio.set_event_loop()</td>
<td>Set the event loop as current via the current policy.</td>
</tr>
<tr>
<td>asyncio.new_event_loop()</td>
<td>Create a new event loop.</td>
</tr>
</tbody>
</table>

Examples

- Using asyncio.get_running_loop().
Event Loop Methods

See also the main documentation section about the event loop methods.

Lifecycle

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.run_until_complete()</code></td>
<td>Run a Future/Task/awaitable until complete.</td>
</tr>
<tr>
<td><code>loop.run_forever()</code></td>
<td>Run the event loop forever.</td>
</tr>
<tr>
<td><code>loop.stop()</code></td>
<td>Stop the event loop.</td>
</tr>
<tr>
<td><code>loop.close()</code></td>
<td>Close the event loop.</td>
</tr>
<tr>
<td><code>loop.is_running()</code></td>
<td>Return True if the event loop is running.</td>
</tr>
<tr>
<td><code>loop.is_closed()</code></td>
<td>Return True if the event loop is closed.</td>
</tr>
<tr>
<td><code>await loop.shutdown_asyncgens()</code></td>
<td>Close asynchronous generators.</td>
</tr>
</tbody>
</table>

Debugging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.set_debug()</code></td>
<td>Enable or disable the debug mode.</td>
</tr>
<tr>
<td><code>loop.get_debug()</code></td>
<td>Get the current debug mode.</td>
</tr>
</tbody>
</table>

Scheduling Callbacks

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.call_soon()</code></td>
<td>Invoke a callback soon.</td>
</tr>
<tr>
<td><code>loop.call_soon_threadsafe()</code></td>
<td>A thread-safe variant of</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>loop.call_soon()</code></td>
<td><em>loop.call_soon().</em></td>
</tr>
<tr>
<td><code>loop.call_later()</code></td>
<td>Invoke a callback <em>after</em> the given time.</td>
</tr>
<tr>
<td><code>loop.call_at()</code></td>
<td>Invoke a callback <em>at</em> the given time.</td>
</tr>
</tbody>
</table>

**Thread/Process Pool**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await loop.run_in_executor()</code></td>
<td>Run a CPU-bound or other blocking function in a <em>concurrent.futures</em> executor.</td>
</tr>
<tr>
<td><code>loop.set_default_executer()</code></td>
<td>Set the default executor for <em>loop.run_in_executor().</em></td>
</tr>
</tbody>
</table>

**Tasks and Futures**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.create_future()</code></td>
<td>Create a <em>Future</em> object.</td>
</tr>
<tr>
<td><code>loop.create_task()</code></td>
<td>Schedule coroutine as a <em>Task</em>.</td>
</tr>
<tr>
<td><code>loop.set_task_factory()</code></td>
<td>Set a factory used by <em>loop.create_task()</em> to create <em>Tasks</em>.</td>
</tr>
<tr>
<td><code>loop.get_task_factory()</code></td>
<td>Get the factory <em>loop.create_task()</em> uses to create <em>Tasks</em>.</td>
</tr>
</tbody>
</table>

**DNS**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await loop.getaddrinfo()</code></td>
<td>Asynchronous version of <em>socket.getaddrinfo().</em></td>
</tr>
</tbody>
</table>
### Networking and IPC

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await loop.getnameinfo()</code></td>
<td>Asynchronous version of <code>socket.getnameinfo()</code></td>
</tr>
<tr>
<td><code>await loop.create_connection()</code></td>
<td>Open a TCP connection.</td>
</tr>
<tr>
<td><code>await loop.create_server()</code></td>
<td>Create a TCP server.</td>
</tr>
<tr>
<td><code>await loop.create_unix_connection()</code></td>
<td>Open a Unix socket connection.</td>
</tr>
<tr>
<td><code>await loop.create_unix_server()</code></td>
<td>Create a Unix socket server.</td>
</tr>
<tr>
<td><code>await loop.connect_accepted_socket()</code></td>
<td>Wrap a socket into a <code>(transport, protocol)</code> pair.</td>
</tr>
<tr>
<td><code>await loop.create_datagram_endpoint()</code></td>
<td>Open a datagram (UDP) connection.</td>
</tr>
<tr>
<td><code>await loop.sendfile()</code></td>
<td>Send a file over a transport.</td>
</tr>
<tr>
<td><code>await loop.start_tls()</code></td>
<td>Upgrade an existing connection to TLS.</td>
</tr>
<tr>
<td><code>await loop.connect_read_pipe()</code></td>
<td>Wrap a read end of a pipe into a <code>(transport, protocol)</code> pair.</td>
</tr>
<tr>
<td><code>await loop.connect_write_pipe()</code></td>
<td>Wrap a write end of a pipe into a <code>(transport, protocol)</code> pair.</td>
</tr>
</tbody>
</table>
Sockets

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>await loop.sock_recv()</code></td>
<td>Receive data from the socket.</td>
</tr>
<tr>
<td><code>await loop.sock_recv_into()</code></td>
<td>Receive data from the socket into a buffer.</td>
</tr>
<tr>
<td><code>await loop.sock_sendall()</code></td>
<td>Send data to the socket.</td>
</tr>
<tr>
<td><code>await loop.sock_connect()</code></td>
<td>Connect the socket.</td>
</tr>
<tr>
<td><code>await loop.sock_accept()</code></td>
<td>Accept a socket connection.</td>
</tr>
<tr>
<td><code>await loop.sock_sendfile()</code></td>
<td>Send a file over the socket.</td>
</tr>
<tr>
<td><code>loop.add_reader()</code></td>
<td>Start watching a file descriptor for read availability.</td>
</tr>
<tr>
<td><code>loop.remove_reader()</code></td>
<td>Stop watching a file descriptor for read availability.</td>
</tr>
<tr>
<td><code>loop.add_writer()</code></td>
<td>Start watching a file descriptor for write availability.</td>
</tr>
<tr>
<td><code>loop.remove_writer()</code></td>
<td>Stop watching a file descriptor for write availability.</td>
</tr>
</tbody>
</table>

Unix Signals

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.add_signal_handler()</code></td>
<td>Add a handler for a signal.</td>
</tr>
<tr>
<td><code>loop.remove_signal_handler()</code></td>
<td>Remove a handler for a signal.</td>
</tr>
</tbody>
</table>

Subprocesses
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.subprocess_exec()</code></td>
<td>Spawn a subprocess.</td>
</tr>
<tr>
<td><code>loop.subprocess_shell()</code></td>
<td>Spawn a subprocess from a shell command.</td>
</tr>
</tbody>
</table>

**Error Handling**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loop.call_exception_handler()</code></td>
<td>Call the exception handler.</td>
</tr>
<tr>
<td><code>loop.set_exception_handler()</code></td>
<td>Set a new exception handler.</td>
</tr>
<tr>
<td><code>loop.get_exception_handler()</code></td>
<td>Get the current exception handler.</td>
</tr>
<tr>
<td><code>loop.default_exception_handler()</code></td>
<td>The default exception handler implementation.</td>
</tr>
</tbody>
</table>

**Examples**

- Using `asyncio.get_event_loop()` and `loop.run_forever()`.
- Using `loop.call_later()`.
- Using `loop.create_connection()` to implement an echo-client.
- Using `loop.create_connection()` to connect a socket.
- Using `add_reader()` to watch an FD for read events.
- Using `loop.add_signal_handler()`.
- Using `loop.subprocess_exec()`.
Transports

All transports implement the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transport.close()</code></td>
<td>Close the transport.</td>
</tr>
<tr>
<td><code>transport.is_closing()</code></td>
<td>Return True if the transport is closing or is closed.</td>
</tr>
<tr>
<td><code>transport.get_extra_info()</code></td>
<td>Request for information about the transport.</td>
</tr>
<tr>
<td><code>transport.set_protocol()</code></td>
<td>Set a new protocol.</td>
</tr>
<tr>
<td><code>transport.get_protocol()</code></td>
<td>Return the current protocol.</td>
</tr>
</tbody>
</table>

Transports that can receive data (TCP and Unix connections, pipes, etc). Returned from methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_read_pipe()`, etc:

**Read Transports**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transport.is_reading()</code></td>
<td>Return True if the transport is receiving.</td>
</tr>
<tr>
<td><code>transport.pause_reading()</code></td>
<td>Pause receiving.</td>
</tr>
<tr>
<td><code>transport.resume_reading()</code></td>
<td>Resume receiving.</td>
</tr>
</tbody>
</table>

Transports that can Send data (TCP and Unix connections, pipes, etc). Returned from methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_write_pipe()`, etc:
## Write Transports

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transport.write()</code></td>
<td>Write data to the transport.</td>
</tr>
<tr>
<td><code>transport.writelines()</code></td>
<td>Write buffers to the transport.</td>
</tr>
<tr>
<td><code>transport.can_write_eof()</code></td>
<td>Return <code>True</code> if the transport supports sending EOF.</td>
</tr>
<tr>
<td><code>transport.write_eof()</code></td>
<td>Close and send EOF after flushing buffered data.</td>
</tr>
<tr>
<td><code>transport.abort()</code></td>
<td>Close the transport immediately.</td>
</tr>
<tr>
<td><code>transport.get_write_buffer_size()</code></td>
<td>Return high and low water marks for write flow control.</td>
</tr>
<tr>
<td><code>transport.set_write_buffer_limits()</code></td>
<td>Set new high and low water marks for write flow control.</td>
</tr>
</tbody>
</table>

Transports returned by `loop.create_datagram_endpoint()`:

## Datagram Transports

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transport.sendto()</code></td>
<td>Send data to the remote peer.</td>
</tr>
<tr>
<td><code>transport.abort()</code></td>
<td>Close the transport immediately.</td>
</tr>
</tbody>
</table>
Low-level transport abstraction over subprocesses. Returned by `loop.subprocess_exec()` and `loop.subprocess_shell()`:

**Subprocess Transports**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transport.get_pid()</code></td>
<td>Return the subprocess process id.</td>
</tr>
<tr>
<td><code>transport.get_pipe_transport()</code></td>
<td>Return the transport for the requested communication pipe (<code>stdin</code>, <code>stdout</code>, or <code>stderr</code>).</td>
</tr>
<tr>
<td><code>transport.get_returncode()</code></td>
<td>Return the subprocess return code.</td>
</tr>
<tr>
<td><code>transport.kill()</code></td>
<td>Kill the subprocess.</td>
</tr>
<tr>
<td><code>transport.send_signal()</code></td>
<td>Send a signal to the subprocess.</td>
</tr>
<tr>
<td><code>transport.terminate()</code></td>
<td>Stop the subprocess.</td>
</tr>
<tr>
<td><code>transport.close()</code></td>
<td>Kill the subprocess and close all pipes.</td>
</tr>
</tbody>
</table>
Protocols

Protocol classes can implement the following **callback methods**:

<table>
<thead>
<tr>
<th>Callback Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection_made()</code></td>
<td>Called when a connection is made.</td>
</tr>
<tr>
<td><code>connection_lost()</code></td>
<td>Called when the connection is lost or closed.</td>
</tr>
<tr>
<td><code>pause_writing()</code></td>
<td>Called when the transport’s buffer goes over the high water mark.</td>
</tr>
<tr>
<td><code>resume_writing()</code></td>
<td>Called when the transport’s buffer drains below the low water mark.</td>
</tr>
</tbody>
</table>

**Streaming Protocols (TCP, Unix Sockets, Pipes)**

<table>
<thead>
<tr>
<th>Callback Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data_received()</code></td>
<td>Called when some data is received.</td>
</tr>
<tr>
<td><code>eof_received()</code></td>
<td>Called when an EOF is received.</td>
</tr>
</tbody>
</table>

**Buffered Streaming Protocols**

<table>
<thead>
<tr>
<th>Callback Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_buffer()</code></td>
<td>Called to allocate a new receive buffer.</td>
</tr>
<tr>
<td><code>buffer_updated()</code></td>
<td>Called when the buffer was updated with the received data.</td>
</tr>
<tr>
<td><code>eof_received()</code></td>
<td>Called when an EOF is received.</td>
</tr>
</tbody>
</table>

**Datagram Protocols**
callback

datagram_received()

Called when a datagram is received.

callback

error_received()

Called when a previous send or receive operation raises an 
OSError.

Subprocess Protocols

callback

pipe_data_received()

Called when the child process writes data into its stdout or 
stderr pipe.

callback

pipe_connection_lost()

Called when one of the pipes communicating with the child 
process is closed.

callback

process_exited()

Called when the child process has exited.
Event Loop Policies

Policies is a low-level mechanism to alter the behavior of functions like `asyncio.get_event_loop()`. See also the main policies section for more details.

**Accessing Policies**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asyncio.get_event_loop_policy()</code></td>
<td>Return the current process-wide policy.</td>
</tr>
<tr>
<td><code>asyncio.set_event_loop_policy()</code></td>
<td>Set a new process-wide policy.</td>
</tr>
<tr>
<td><code>AbstractEventLoopPolicy</code></td>
<td>Base class for policy objects.</td>
</tr>
</tbody>
</table>

Developing with asyncio

Asynchronous programming is different from classic “sequential” programming.

This page lists common mistakes and traps and explains how to avoid them.
Debug Mode

By default asyncio runs in production mode. In order to ease the development asyncio has a *debug mode*.

There are several ways to enable asyncio debug mode:

- Setting the `PYTHONASYNCIODEBUG` environment variable to 1.
- Using the `-X dev` Python command line option.
- Passing `debug=True` to `asyncio.run()`.
- Calling `loop.set_debug()`.

In addition to enabling the debug mode, consider also:

- setting the log level of the asyncio logger to `logging.DEBUG`, for example the following snippet of code can be run at startup of the application:

  ```python
  logging.basicConfig(level=logging.DEBUG)
  ```

- configuring the `warnings` module to display `ResourceWarning` warnings. One way of doing that is by using the `-W default` command line option.

When the debug mode is enabled:

- asyncio checks for coroutines that were not awaited and logs them; this mitigates the “forgotten await” pitfall.
- Many non-threadsafe asyncio APIs (such as `loop.callSoon()` and `loop.callAt()` methods) raise an exception if they are called from a wrong thread.
- The execution time of the I/O selector is logged if it takes too long to perform an I/O operation.
• Callbacks taking longer than 100ms are logged. The `loop.slow_callback_duration` attribute can be used to set the minimum execution duration in seconds that is considered "slow".
Concurrency and Multithreading

An event loop runs in a thread (typically the main thread) and executes all callbacks and Tasks in its thread. While a Task is running in the event loop, no other Tasks can run in the same thread. When a Task executes an `await` expression, the running Task gets suspended, and the event loop executes the next Task.

To schedule a callback from a different OS thread, the `loop.call_soon_threadsafe()` method should be used. Example:

```python
loop.call_soon_threadsafe(callback, *args)
```

Almost all asyncio objects are not thread safe, which is typically not a problem unless there is code that works with them from outside of a Task or a callback. If there's a need for such code to call a low-level asyncio API, the `loop.call_soon_threadsafe()` method should be used, e.g.:

```python
loop.call_soon_threadsafe(fut.cancel)
```

To schedule a coroutine object from a different OS thread, the `run_coroutine_threadsafe()` function should be used. It returns a `concurrent.futures.Future` to access the result:

```python
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:
future = asyncio.run_coroutine_threadsafe(coro_func()
To handle signals and to execute subprocesses, the event loop must be run in the main thread.

The `loop.run_in_executor()` method can be used with a `concurrent.futures.ThreadPoolExecutor` to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.
Running Blocking Code

Blocking (CPU-bound) code should not be called directly. For example, if a function performs a CPU-intensive calculation for 1 second, all concurrent asyncio Tasks and IO operations would be delayed by 1 second.

An executor can be used to run a task in a different thread or even in a different process to avoid blocking block the OS thread with the event loop. See the `loop.run_in_executor()` method for more details.
Logging

asyncio uses the `logging` module and all logging is performed via the "asyncio" logger.

The default log level is `logging.INFO`, which can be easily adjusted:

```python
logging.getLogger("asyncio").setLevel(logging.WARN)
```
Detect never-awaited coroutines

When a coroutine function is called, but not awaited (e.g. `coro()`) instead of `await coro()` or the coroutine is not scheduled with `asyncio.create_task()`, asyncio will emit a `RuntimeWarning`:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Output:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
    test()
```

Output in debug mode:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited

Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  <..>  

  File "../t.py", line 7, in main
  test()
  test()
```

The usual fix is to either await the coroutine or call the `asyncio.create_task()` function:

```python
async def main():
    await test()
```
Detect never-retrieved exceptions

If a `Future.set_exception()` is called but the Future object is never awaited on, the exception would never be propagated to the user code. In this case, asyncio would emit a log message when the Future object is garbage collected.

Example of an unhandled exception:

```python
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

Output:

```
Task exception was never retrieved
future: Task finished coro=<bug() done, defined at
    exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode to get the traceback where the task was created:
asyncio.run(main(), debug=True)

Output in debug mode:

Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at
timeout_exception=Exception('not consumed') created at a
source_traceback: Object created at (most recent call)
File ".../t.py", line 9, in <module>
  asyncio.run(main(), debug=True)
<..>

Traceback (most recent call last):
  File ".../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
socket — Low-level networking interface

Source code: Lib/socket.py

This module provides access to the BSD socket interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also:

Module socketserver
Classes that simplify writing network servers.

Module ssl
A TLS/SSL wrapper for socket objects.
Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an AF_UNIX socket bound to a file system node is represented as a string, using the file system encoding and the 'surrogateescape' error handler (see PEP 383). An address in Linux's abstract namespace is returned as a bytes-like object with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Changed in version 3.3: Previously, AF_UNIX socket paths were assumed to use UTF-8 encoding.

  Changed in version 3.5: Writable bytes-like object is now accepted.

- A pair (host, port) is used for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like 'daring.cwi.nl' or an IPv4 address like '100.50.200.5', and port is an integer.

  For IPv4 addresses, two special forms are accepted instead...
of a host address: '' represents INADDR_ANY, which is used to bind to all interfaces, and the string '<broadcast>' represents INADDR_BROADCAST. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

- For AF_INET6 address family, a four-tuple (host, port, flowinfo, scopeid) is used, where flowinfo and scopeid represent the sin6_flowinfo and sin6_scope_id members in struct sockaddr_in6 in C. For socket module methods, flowinfo and scopeid can be omitted just for backward compatibility. Note, however, omission of scopeid can cause problems in manipulating scoped IPv6 addresses.

  Changed in version 3.7: For multicast addresses (with scopeid meaningful) address may not contain %scope (or zone id) part. This information is superfluous and may be safely omitted (recommended).

- AF_NETLINK sockets are represented as pairs (pid, groups).

- Linux-only support for TIPC is available using the AF_TIPC address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (addr_type, v1, v2, v3 [, scope]), where:
  - `addr_type` is one of TIPC_ADDR_NAMESEQ, TIPC_ADDR_NAME, or TIPC_ADDR_ID.
  - `scope` is one of TIPC_ZONE_SCOPE,
TIPC_CLUSTER_SCOPE, and TIPC_NODE_SCOPE.

- If `addr_type` is TIPC_ADDR_NAME, then `v1` is the server type, `v2` is the port identifier, and `v3` should be 0.

- If `addr_type` is TIPC_ADDR_NAMESEQ, then `v1` is the server type, `v2` is the lower port number, and `v3` is the upper port number.

- If `addr_type` is TIPC_ADDR_ID, then `v1` is the node, `v2` is the reference, and `v3` should be set to 0.

- A tuple `(interface,)` is used for the AF_CAN address family, where `interface` is a string representing a network interface name like `'can0'`. The network interface name `' '` can be used to receive packets from all network interfaces of this family.

  - CAN_ISOTP protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).

- A string or a tuple `(id, unit)` is used for the SYSPROTO_CONTROL protocol of the PF_SYSTEM family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  *New in version 3.3.*

- AF_BLUETOOTH supports the following protocols and address formats:

  - BTPROTO_L2CAP accepts `(bdaddr, psm)` where `bdaddr`
is the Bluetooth address as a string and `psm` is an integer.

- `BPROTO_RFCOMM` accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.

- `BPROTO_HCI` accepts `(device_id,)` where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

  Changed in version 3.2: NetBSD and DragonFlyBSD support added.

- `BPROTO_SCO` accepts `bdaddr` where `bdaddr` is a `bytes` object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.

- `AF_ALG` is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:

  - `type` is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
  - `name` is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
  - `feat` and `mask` are unsigned 32bit integers.

  **Availability:** Linux 2.6.38, some algorithm types require more recent Kernels.
New in version 3.6.

- **AF_VSOCK** allows communication between virtual machines and their hosts. The sockets are represented as a \((\text{CID, port})\) tuple where the context ID or CID and port are integers.

**Availability:** Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

New in version 3.7.

- **AF_PACKET** is a low-level interface directly to network devices. The packets are represented by the tuple \((\text{ifname, proto[, pkttype[, hatype[, addr]]]}))\) where:
  - **ifname** - String specifying the device name.
  - **proto** - An in network-byte-order integer specifying the Ethernet protocol number.
  - **pkttype** - Optional integer specifying the packet type:
    - **PACKET_HOST** (the default) - Packet addressed to the local host.
    - **PACKET_BROADCAST** - Physical-layer broadcast packet.
    - **PACKET_MULTIHOST** - Packet sent to a physical-layer multicast address.
    - **PACKET_OTHERHOST** - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - **PACKET_OUTGOING** - Packet originating from the local host that is looped back to a packet socket.
  - **hatype** - Optional integer specifying the ARP hardware address type.
  - **addr** - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.
If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise *OSERROR* or one of its subclasses (they used to raise *socket.error*).

Non-blocking mode is supported through *setblocking()* . A generalization of this based on timeouts is supported through *settimeout()*.
Module contents

The module `socket` exports the following elements.

Exceptions

`exception` `socket.error`

A deprecated alias of `OSError`.

*Changed in version 3.3:* Following PEP 3151, this class was made an alias of `OSError`.

`exception` `socket.herror`

A subclass of `OSError`, this exception is raised for address-related errors, i.e. for functions that use `h_errno` in the POSIX C API, including `gethostbyname_ex()` and `gethostbyaddr()`. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. `h_errno` is a numeric value, while `string` represents the description of `h_errno`, as returned by the `hstrerror()` C function.

*Changed in version 3.3:* This class was made a subclass of `OSError`.

`exception` `socket.gaierror`

A subclass of `OSError`, this exception is raised for address-related errors by `getaddrinfo()` and `getnameinfo()`. The accompanying value is a pair `(error, string)` representing an error returned by a library call. `string` represents the description of `error`, as returned by the `gai_strerror()` C function. The numeric `error` value will match one of the EAI_* constants defined
in this module.

*Changed in version 3.3:* This class was made a subclass of `OSError`.

*exception* `socket.timeout`

A subclass of `OSError`, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()` (or implicitly through `setdefaulttimeout()`). The accompanying value is a string whose value is currently always “timed out”.

*Changed in version 3.3:* This class was made a subclass of `OSError`.

**Constants**

The `AF_*` and `SOCK_*` constants are now `AddressFamily` and `SocketKind` `IntEnum` collections.

*New in version 3.4.*

`socket.AF_UNIX`

`socket.AF_INET`

`socket.AF_INET6`

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

`socket.SOCK_STREAM`

`socket.SOCK_DGRAM`

`socket.SOCK_RAW`
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to `socket()`. More constants may be available depending on the system. (Only **SOCK_STREAM** and **SOCK_DGRAM** appear to be generally useful.)

socket.**SOCK_CLOEXEC**
socket.**SOCK_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

**See also:** [Secure File Descriptor Handling](#) for a more thorough explanation.

**Availability:** Linux >= 2.6.27.

*New in version 3.2.*

**SO_***
socket.**SOMAXCONN**
**MSG_***
**SOL_***
**SCM_***
**IPPROTO_***
**IPPORT_***
**INADDR_***
**IP_***
**IPV6_***
**EAI_***
**AI_***
**NI_***
TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

*Changed in version 3.6:* `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

*Changed in version 3.6.5:* On Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` appear if run-time Windows supports.

*Changed in version 3.7:* `TCP_NOTSENT_LOWAT` was added.

On Windows, `TCP_KEEPIDLE`, `TCP_KEEPINTVL` appear if run-time Windows supports.

socket.**AF_CAN**

socket.**PF_CAN**

**SOL_CAN_***

**CAN_***

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

**Availability:** Linux >= 2.6.25.

*New in version 3.3.*

socket.**CAN_BCM**
CAN_BCM_*
CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

New in version 3.4.

socket.CAN_RAW_FD_FRAMES
Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you one must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability: Linux >= 3.6.

New in version 3.5.

socket.CAN_ISOTP
CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Availability: Linux >= 2.6.25.

New in version 3.7.

socket.AF_PACKET
socket.PF_PACKET
PACKET_*
Many constants of these forms, documented in the Linux
documentation, are also defined in the socket module.

**Availability:** Linux >= 2.2.

socket.AF_RDS
socket.PF_RDS
socket.SOL_RDS
RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

**Availability:** Linux >= 2.6.30.

*New in version 3.3.*

socket.SIO_RCVALL
socket.SIO_KEEPALIVE_VALS
socket.SIO_LOOPBACK_FAST_PATH
RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as arguments to the ioctl() method of socket objects.

*Changed in version 3.6:* SIO_LOOPBACK_FAST_PATH was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.AF_ALG
socket.SOL_ALG
ALG_*

Constants for Linux Kernel cryptography.

**Availability:** Linux >= 2.6.38.
New in version 3.6.

socket.AF_VSOCK
socket.IOCTL_VM_SOCKETS_GET_LOCAL_CID
VMADDR*
SO_VM*

Constants for Linux host/guest communication.

Availability: Linux >= 4.8.

New in version 3.7.

socket.AF_LINK

Availability: BSD, OSX.

New in version 3.4.

socket.has_ipv6

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.BDADDR_ANY
socket.BDADDR_LOCAL

These are string constants containing Bluetooth addresses with special meanings. For example, BDADDR_ANY can be used to indicate any address when specifying the binding socket with BTPROTO_RFCOMM.

socket.HCI_FILTER
socket.HCI_TIME_STAMP
socket.HCI_DATA_DIR

For use with BTPROTO_HCI. HCI_FILTER is not available for NetBSD or DragonFlyBSD. HCI_TIME_STAMP and
**HCI_DATA_DIR** are not available for FreeBSD, NetBSD, or DragonFlyBSD.

## Functions

### Creating sockets

The following functions all create socket objects.

```python
socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

Create a new socket using the given address family, socket type and protocol number. The address family should be **AF_INET** (the default), **AF_INET6**, **AF_UNIX**, **AF_CAN**, **AF_PACKET**, or **AF_RDS**. The socket type should be **SOCK_STREAM** (the default), **SOCK_DGRAM**, **SOCK_RAW** or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is **AF_CAN** the protocol should be one of **CAN_RAW**, **CAN_BCM** or **CAN_ISOTP**.

If `fileno` is specified, the values for `family`, `type`, and `proto` are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit `family`, `type`, or `proto` arguments. This only affects how Python represents e.g. the return value of `socket.getpeername()` but not the actual OS resource. Unlike `socket.fromfd()`, `fileno` will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

The newly created socket is **non-inheritable**.

*Changed in version 3.3*: The **AF_CAN** family was added. The
AF_RDS family was added.

*Changed in version 3.4:* The CAN_BCM protocol was added.

*Changed in version 3.4:* The returned socket is now non-inheritable.

*Changed in version 3.7:* The CAN_ISOTP protocol was added.

*Changed in version 3.7:* When `SOCK_NONBLOCK` or `SOCK_CLOEXEC` bit flags are applied to `type` they are cleared, and `socket.type` will not reflect them. They are still passed to the underlying system `socket()` call. Therefore::

```python
sock = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM |
    socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

**socket.socketpair([family[, type[, proto]]])**

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the `socket()` function above. The default family is `AF_UNIX` if defined on the platform; otherwise, the default is `AF_INET`.

The newly created sockets are non-inheritable.

*Changed in version 3.2:* The returned socket objects now support the whole socket API, rather than a subset.
*Changed in version 3.4:* The returned sockets are now non-inheritable.

*Changed in version 3.5:* Windows support added.

```python
socket.create_connection(address[, timeout[, source_address]])
```

Connect to a TCP service listening on the Internet `address` (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than `socket.connect()`: if `host` is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional `timeout` parameter will set the timeout on the socket instance before attempting to connect. If no `timeout` is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used.

If supplied, `source_address` must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If `host` or `port` are "" or 0 respectively the OS default behavior will be used.

*Changed in version 3.2:* `source_address` was added.

```python
socket.fromfd(fd, family, type, proto=0)
```

Duplicate the file descriptor `fd` (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the `socket()` function above. The file descriptor should refer to a
socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is non-inheritable.

*Changed in version 3.4:* The returned socket is now non-inheritable.

**socket.fromshare(data)**

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

**Availability:** Windows.

*New in version 3.3.*

**socket.SocketType**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

**Other functions**

The `socket` module also offers various network-related services:

**socket.close(fd)**

Close a socket file descriptor. This is like `os.close()`, but for sockets. On some platforms (most noticeable Windows) `os.close()` does not work for socket file descriptors.
socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. host is a domain name, a string representation of an IPv4/v6 address or None. port is a string service name such as 'http', a numeric port number or None. By passing None as the value of host and port, you can pass NULL to the underlying C API.

The family, type and proto arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The flags argument can be one or several of the AI_* constants, and will influence how results are computed and returned. For example, AI_NUMERICHOST will disable domain name resolution and will raise an error if host is a domain name.

The function returns a list of 5-tuples with the following structure:

(family, type, proto, canonname, sockaddr)

In these tuples, family, type, proto are all integers and are meant to be passed to the socket() function. canonname will be a string representing the canonical name of the host if AI_CANONNAME is part of the flags argument; else canonname will be empty. sockaddr is a tuple describing a socket address, whose format depends on the returned family (a (address, port) 2-tuple for AF_INET, a (address, port, flow info, scope id) 4-tuple for AF_INET6), and is meant to be passed to the
socket.connect() method.

The following example fetches address information for a hypothetical TCP connection to example.org on port 80 (results may differ on your system if IPv6 isn’t enabled):

```python
>>> socket.getaddrinfo("example.org", 80, proto=socket.AF_INET6, 0, 0, 0, 0, ('2606:2800:220:1:248:1893:25c8:1946', 80))
```

`Changed in version 3.2:` parameters can now be passed using keyword arguments.

`Changed in version 3.7:` for IPv6 multicast addresses, string representing an address will not contain `scope` part.

socket.getfqdn([name])

Return a fully qualified domain name for `name`. If `name` is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by `gethostname()` is returned.

socket.gethostbyname(hostname)

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface.
gethostbyname() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

socket.gethostbyname_ex(hostname)
Translate a host name to IPv4 address format, extended interface. Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). gethostbyname_ex() does not support IPv6 name resolution, and getaddrinfo() should be used instead for IPv4/v6 dual stack support.

socket.gethostname()
Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Note: gethostname() doesn't always return the fully qualified domain name; use getfqdn() for that.

socket.gethostbyaddr(ip_address)
Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function getfqdn(). gethostbyaddr() supports both IPv4 and IPv6.
socket.**getnameinfo**(sockaddr, flags)

Translate a socket address `sockaddr` into a 2-tuple `(host, port)`. Depending on the settings of `flags`, the result can contain a fully-qualified domain name or numeric address representation in `host`. Similarly, `port` can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope` is appended to the host part if `sockaddr` contains meaningful `scopeid`. Usually this happens for multicast addresses.

socket.**getprotobynamel**(protocolname)

Translate an Internet protocol name (for example, 'icmp') to a constant suitable for passing as the (optional) third argument to the `socket()` function. This is usually only needed for sockets opened in “raw” mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

socket.**getservbyname**(servicename[, protocolname])

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket.**getservbyport**(port[, protocolname])

Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be 'tcp' or 'udp', otherwise any protocol will match.

socket.**ntohl**(x)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte
order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.\texttt{ntohs}(x)
Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

\textit{Deprecated since version 3.7}: In case \textit{x} does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

socket.\texttt{htonl}(x)
Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.\texttt{htons}(x)
Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

\textit{Deprecated since version 3.7}: In case \textit{x} does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.
socket.inet_aton(ip_string)
Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page inet(3) for details.

If the IPv4 address string passed to this function is invalid, OSError will be raised. Note that exactly what is valid depends on the underlying C implementation of inet_aton().

inet_aton() does not support IPv6, and inet_pton() should be used instead for IPv4/v6 dual stack support.

socket.inet_ntoa(packed_ip)
Convert a 32-bit packed IPv4 address (a bytes-like object four bytes in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, OSError will be raised. inet_ntoa() does not support IPv6, and inet_ntop() should be used instead for IPv4/v6 dual stack support.

Changed in version 3.5: Writable bytes-like object is now accepted.
socket.inet_pton(address_family, ip_string)

Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the IP address string `ip_string` is invalid, `OSError` will be raised. Note that exactly what is valid depends on both the value of `address_family` and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added

socket.inet_ntop(address_family, packed_ip)

Convert a packed IP address (a bytes-like object of some number of bytes) to its standard, family-specific string representation (for example, '7.10.0.5' or '5aef:2b::8'). `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the bytes object `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. `OSError` is raised for errors from the call to `inet_ntop()`.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added
Changed in version 3.5: Writable bytes-like object is now accepted.

socket.\texttt{CMSG\_LEN}(length)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given length. This value can often be used as the buffer size for \texttt{recvmsg()} to receive a single item of ancillary data, but RFC 3542 requires portable applications to use \texttt{CMSG\_SPACE()} and thus include space for padding, even when the item will be the last in the buffer. Raises \texttt{OverflowError} if length is outside the permissible range of values.

\textbf{Availability}: most Unix platforms, possibly others.

New in version 3.3.

socket.\texttt{CMSG\_SPACE}(length)

Return the buffer size needed for \texttt{recvmsg()} to receive an ancillary data item with associated data of the given length, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the \texttt{CMSG\_SPACE()} values for their associated data lengths. Raises \texttt{OverflowError} if length is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

\textbf{Availability}: most Unix platforms, possibly others.

New in version 3.3.
socket.getdefaulttimeout()
Return the default timeout in seconds (float) for new socket objects. A value of None indicates that new socket objects have no timeout. When the socket module is first imported, the default is None.

socket.setdefaulttimeout(timeout)
Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is None. See settimeout() for possible values and their respective meanings.

socket.sethostname(name)
Set the machine’s hostname to name. This will raise an OSError if you don’t have enough rights.

Availability: Unix.
New in version 3.3.

socket.if_nameindex()
Return a list of network interface information (index int, name string) tuples. OSError if the system call fails.

Availability: Unix.
New in version 3.3.

socket.if_nametoindex(if_name)
Return a network interface index number corresponding to an interface name. OSError if no interface with the given name exists.

Availability: Unix.
socket.`if_indextoname(if_index)`

Return a network interface name corresponding to an interface index number. `OSError` if no interface with the given index exists.

**Availability**: Unix.

*New in version 3.3.*
Socket Objects

Socket objects have the following methods. Except for \texttt{makefile()}, these correspond to Unix system calls applicable to sockets.

*Changed in version 3.2:* Support for the \texttt{context manager} protocol was added. Exiting the context manager is equivalent to calling \texttt{close()}.

\texttt{socket.accept()}

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair \((\text{conn, address})\) where \texttt{conn} is a \textit{new} socket object usable to send and receive data on the connection, and \texttt{address} is the address bound to the socket on the other end of the connection.

The newly created socket is \textit{non-inheritable}.

*Changed in version 3.4:* The socket is now non-inheritable.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an \texttt{InterruptedError} exception (see \texttt{PEP 475} for the rationale).

\texttt{socket.bind(address)}

Bind the socket to \texttt{address}. The socket must not already be bound. (The format of \texttt{address} depends on the address family — see above.)

\texttt{socket.close()}

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from \texttt{makefile()}
are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

*Changed in version 3.6:* `OSError` is now raised if an error occurs when the underlying `close()` call is made.

Note: `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

`socket.connect(address)`

Connect to a remote socket at `address`. (The format of `address` depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `socket.timeout` on timeout, if the signal handler doesn’t raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

*Changed in version 3.5:* The method now waits until the connection completes instead of raising an `InterruptedError` exception if the connection is interrupted by a signal, the signal handler doesn’t raise an exception and the socket is blocking or has a timeout (see
the PEP 475 for the rationale).

socket.connect_ex(address)
Like connect(address), but return an error indicator instead of raising an exception for errors returned by the C-level connect() call (other problems, such as “host not found,” can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the errno variable. This is useful to support, for example, asynchronous connects.

socket.detach()
Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

New in version 3.2.

socket.dup()
Duplicate the socket.

The newly created socket is non-inheritable.

Changed in version 3.4: The socket is now non-inheritable.

socket.fileno()
Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with select.select().

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as os.fdopen()). Unix does not have this limitation.

socket.get_inheritable()
Get the **inheritable flag** of the socket's file descriptor or socket's handle: True if the socket can be inherited in child processes, False if it cannot.

*New in version 3.4.*

**socket.getpeernam**e()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

**socket.getsockname**()

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

**socket.getsockopt**(level, optname[, buflen])

Return the value of the given socket option (see the Unix man page `getsockopt(2)`). The needed symbolic constants (SO_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as byte strings).

**socket.getblocking**()

Return True if socket is in blocking mode, False if in non-blocking.
This is equivalent to checking `socket.gettimeout() == 0`.

*New in version 3.7.*

**socket.gettimeout()**

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to `setblocking()` or `settimeout()`.

**socket.ioctl(control, option)**

*Platform: Windows*

The `ioctl()` method is a limited interface to the WSAIoctl system interface. Please refer to the *Win32 documentation* for more information.

On other platforms, the generic `fcntl.fcntl()` and `fcntl.ioctl()` functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

*Changed in version 3.6:* `SIO_LOOPBACK_FAST_PATH` was added.

**socket.listen([backlog])**

Enable a server to accept connections. If `backlog` is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.
Changed in version 3.5: The backlog parameter is now optional.

socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). These arguments are interpreted the same way as by the built-in open() function, except the only supported mode values are 'r' (default), 'w' and 'b'.

The socket must be in blocking mode; it can have a timeout, but the file object’s internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by makefile() won’t close the original socket unless all other file objects have been closed and socket.close() has been called on the socket object.

Note: On Windows, the file-like object created by makefile() cannot be used where a file object with a file descriptor is expected, such as the stream arguments of subprocess.Popen().

socket.recv(bufsize[, flags])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

Note: For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for
example, 4096.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

```
socket.recvfrom(bufsize[, flags])
```

Receive data from the socket. The return value is a pair `(bytes, address)` where `bytes` is a bytes object representing the data received and `address` is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument `flags`; it defaults to zero. (The format of `address` depends on the address family — see above.)

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

*Changed in version 3.7:* For multicast IPv6 address, first item of `address` does not contain `%scope` part anymore. In order to get full IPv6 address use `getnameinfo()`.

```
socket.recvmsg(bufsize[, ancbufsize[, flags]])
```

Receive normal data (up to `bufsize` bytes) and ancillary data from the socket. The `ancbufsize` argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The `flags` argument defaults to 0 and has
the same meaning as for `recv()`.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The `data` item is a `bytes` object holding the non-ancillary data received. The `ancdata` item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: `cmsg_level` and `cmsg_type` are integers specifying the protocol level and protocol-specific type respectively, and `cmsg_data` is a `bytes` object holding the associated data. The `msg_flags` item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, `address` is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where `fds` is a `bytes` object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.
On systems which support the SCM_RIGHTS mechanism, the following function will receive up to \textit{maxfds} file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also \texttt{sendmsg()}. 

```python
import socket, array
def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")  # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen)
    for cmsg_level, cmsg_type, cmsg_data in ancdata
        if (cmsg_level == socket.SOL_SOCKET and
c            # Append data, ignoring any truncated
            fds.fromstring(cmsg_data[:len(cmsg_data)])
    return msg, list(fds)
```

\textbf{Availability:} most Unix platforms, possibly others.

\textit{New in version 3.3.}

\textit{Changed in version 3.5:} If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an \texttt{InterruptedError} exception (see \texttt{PEP 475} for the rationale).

\texttt{socket.recvmsg_into(buffers[, ancbufsize[, flags]])}

Receive normal data and ancillary data from the socket, behaving as \texttt{recvmsg()} would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The \texttt{buffers} argument must be an iterable of objects that export writable buffers (e.g. \texttt{bytearray} objects); these will be filled with successive chunks of the non-ancillary data until it has all been
written or there are no more buffers. The operating system may set a limit (sysconf() value SC_IOV_MAX) on the number of buffers that can be used. The ancbufsize and flags arguments have the same meaning as for recvmsg().

The return value is a 4-tuple: \((nbytes, \text{ancdata}, \text{msg\_flags}, \text{address})\), where \(nbytes\) is the total number of bytes of non-ancillary data written into the buffers, and \(\text{ancdata}\), \(\text{msg\_flags}\) and \(\text{address}\) are the same as for recvmsg().

Example:

```python
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'-----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

**Availability**: most Unix platforms, possibly others.

*New in version 3.3.*

socket.recvfrom_into(buffer[, nbytes[, flags]])

Receive data from the socket, writing it into \(\text{buffer}\) instead of creating a new bytestring. The return value is a pair \((nbytes, \text{address})\) where \(nbytes\) is the number of bytes received and \(\text{address}\) is the address of the socket sending the data. See the
Unix manual page `recv(2)` for the meaning of the optional argument `flags`; it defaults to zero. (The format of `address` depends on the address family — see above.)

socket. `recv_into(buffer[, nbytes[, flags]])`  
Receive up to `nbytes` bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If `nbytes` is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page `recv(2)` for the meaning of the optional argument `flags`; it defaults to zero.

socket. `send(bytes[, flags])`  
Send data to the socket. The socket must be connected to a remote socket. The optional `flags` argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the Socket Programming HOWTO.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

socket. `sendall(bytes[, flags])`  
Send data to the socket. The socket must be connected to a remote socket. The optional `flags` argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from `bytes` until either all data has been sent or an error
occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

*Changed in version 3.5:* The socket timeout is no more reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

```python
socket.sendto(bytes, address)
```

`socket.sendto(bytes, flags, address)`

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by `address`. The optional `flags` argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of `address` depends on the address family — see above.)

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

```python
socket.sendmsg(buffers[, ancdata[, flags[, address]]]])
```

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The `buffers` argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`)
on the number of buffers that can be used. The `ancdata` argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where `cmsg_level` and `cmsg_type` are integers specifying the protocol level and protocol-specific type respectively, and `cmsg_data` is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `CMSG_SPACE()`) might support sending only one control message per call. The `flags` argument defaults to 0 and has the same meaning as for `send()`. If `address` is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors `fds` over an `AF_UNIX` socket, on systems which support the SCM_RIGHTS mechanism. See also `recvmsg()`.

```python
import socket, array
def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, 0, 0)])
```

**Availability:** most Unix platforms, possibly others.

*New in version 3.3.*

*Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see PEP 475 for the rationale).

```python
socket.sendmsg_afalg([msg,] *, op[, iv[, assoclen[, flags]]]])
```
Specialized version of `sendmsg()` for AF_ALG socket. Set mode, IV, AEAD associated data length and flags for AF_ALG socket.

**Availability:** Linux >= 2.6.38.

*New in version 3.6.*

`socket.sendfile(file, offset=0, count=None)`

Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. `file` must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or `file` is not a regular file `send()` will be used instead. `offset` tells from where to start reading the file. If specified, `count` is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

*New in version 3.5.*

`socket.set_inheritable(inheritable)`

Set the inheritable flag of the socket's file descriptor or socket's handle.

*New in version 3.4.*

`socket.setblocking(flag)`

Set blocking or non-blocking mode of the socket: if `flag` is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:
• `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
• `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

*Changed in version 3.7:* The method no longer applies `SOCK_NONBLOCK` flag on `socket.type`.

**socket.settimeout(value)**

Set a timeout on blocking socket operations. The `value` argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a `timeout` exception if the timeout period `value` has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the notes on socket timeouts.

*Changed in version 3.7:* The method no longer toggles `SOCK_NONBLOCK` flag on `socket.type`.

**socket.setsockopt(level, optname, value: int)**

**socket.setsockopt(level, optname, value: buffer)**

**socket.setsockopt(level, optname, None, optlen: int)**

Set the value of the given socket option (see the Unix manual page `setsockopt(2)`). The needed symbolic constants are defined in the `socket` module (SO_* etc.). The value can be an integer, `None` or a bytes-like object representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see
the optional built-in module `struct` for a way to encode C structures as bytestrings). When value is set to `None`, optlen argument is required. It’s equivalent to call setsockopt C function with optval=NULL and optlen=optlen.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

*Changed in version 3.6:* setsockopt(level, optname, None, optlen: int) form added.

socket. `shutdown`(how)
Shut down one or both halves of the connection. If `how` is SHUT_RD, further receives are disallowed. If `how` is SHUT_WR, further sends are disallowed. If `how` is SHUT_RDWR, further sends and receives are disallowed.

socket. `share`(process_id)
Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with `process_id`. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

*Availability:* Windows.

*New in version 3.3.*

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without `flags` argument instead.

Socket objects also have these (read-only) attributes that correspond
to the values given to the `socket` constructor.

socket. **family**
   The socket family.

socket. **type**
   The socket type.

socket. **proto**
   The socket protocol.
Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- In **blocking mode**, operations block until complete or the system returns an error (such as connection timed out).
- In **non-blocking mode**, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the `select` can be used to know when and whether a socket is available for reading or writing.
- In **timeout mode**, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

**Note:** At the operating system level, sockets in **timeout mode** are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the connect method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.
Timeouts and the accept method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.
Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()/recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```python
# Echo server program
import socket

HOST = ''  # Symbolic name meaning all available interfaces
PORT = 50007  # Arbitrary non-privileged port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```python
# Echo client program
import socket

HOST = 'daring.cwi.nl'  # The remote host
```
PORT = 50007 # The same port as used by

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

# Echo server program
import socket
import sys

HOST = None # Symbolic name meaning all available interfaces
PORT = 50007 # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM, 0, socket.AI_UNSPEC):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
```python
s = None
continue
break
if s is None:
    print('could not open socket')
sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
```

---

```python
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'  # The remote host
PORT = 50007           # The same port as used by
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                               socket.SOCK_STREAM):
    try:
        s = socket.socket(*res[0], *res[1], *res[2])
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(res[4])
    except OSError as msg:
        s.close()
        s = None
        continue
break
if s is None:
```
print('could not open socket')
sys.exit(1)

with s:
s.sendall(b'Hello, world')
data = s.recv(1024)
print('Received', repr(data))

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```python
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use
CAN with the broadcast manager protocol instead, open a socket with:

```python
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_RAW)
```

After binding (CAN_RAW) or connecting (CAN_BCM) the socket, you can use the `socket.send()` and the `socket.recv()` operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```python
import socket
import struct

# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])

# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)
```
```python
print('Received: can_id=%x, can_dlc=%x, data=%s')

try:
    s.send(cf)
except OSError:
    print('Error sending CAN frame')

try:
    s.send(build_can_frame(0x01, b'\x01\x02\x03'))
except OSError:
    print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSErr0r: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can’t be immediately reused.

There is a `socket` flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```python
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

**See also:** For an introduction to socket programming (in C), see the following papers:
- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al.,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.
ssl — TLS/SSL wrapper for socket objects

Source code: Lib/ssl.py

This module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

**Note:** Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

**Warning:** Don’t use this module without reading the Security considerations. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the ssl module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the “See Also” section at the bottom.

This module provides a class, `ssl.SSLSocket`, which is derived from the `socket.socket` type, and provides a socket-like wrapper that
also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as getpeercert(), which retrieves the certificate of the other side of the connection, and cipher(), which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the `ssl.SSLContext` class helps manage settings and certificates, which can then be inherited by SSL sockets created through the `SSLContext.wrap_socket()` method.

*Changed in version 3.5.3:* Updated to support linking with OpenSSL 1.1.0

*Changed in version 3.6:* OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.
Functions, Constants, and Exceptions

Socket creation

Since Python 3.2 and 2.7.9, it is recommended to use the `SSLContext.wrap_socket()` of an `SSLContext` instance to wrap sockets as `SSLSocket` objects. The helper functions `create_default_context()` returns a new context with secure default settings. The old `wrap_socket()` function is deprecated since it is both inefficient and has no support for server name indication (SNI) and hostname matching.

Client socket example with default context and IPv4/IPv6 dual stack:

```python
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname)
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```python
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    with context.wrap_socket(sock, server_hostname=hostname)
```
print(ssock.version())

Server socket example listening on localhost IPv4:

```python
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True)
        conn, addr = ssock.accept()
```

Context creation

A convenience function helps create `SSLContext` objects for common purposes.

```python
ssl.create_default_context(purpose=Purpose.SERVER_AUTH, cafile=None, capath=None, cadata=None)
```

Return a new `SSLContext` object with default settings for the given `purpose`. The settings are chosen by the `ssl` module, and usually represent a higher security level than when calling the `SSLContext` constructor directly.

`cafile`, `capath`, `cadata` represent optional CA certificates to trust for certificate verification, as in `SSLContext.load_verify_locations()`. If all three are `None`, this function can choose to trust the system's default CA certificates instead.
The settings are: PROTOCOL_TLS, OP_NO_SSLv2, and OP_NO_SSLv3 with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing SERVER_AUTH as purpose sets verify_mode to CERT_REQUIRED and either loads CA certificates (when at least one of cafile, capath or cadata is given) or uses SSLContext.load_default_certs() to load default CA certificates.

**Note:** The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.
If your application needs specific settings, you should create a SSLContext and apply the settings yourself.

**Note:** If you find that when certain older clients or servers attempt to connect with a SSLContext created by this function that they get an error stating “Protocol or cipher suite mismatch”, it may be that they only support SSL3.0 which this function excludes using the OP_NO_SSLv3. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```python
ctx = ssl.create_default_context(Purpose.CLIENT
ctx.options &= ~ssl.OP_NO_SSLv3
```

*New in version 3.4.*

*Changed in version 3.4.4:* RC4 was dropped from the default cipher string.
Changed in version 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Exceptions

`exception ssl.SSLError`

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of `OSError`. The error code and message of `SSLError` instances are provided by the OpenSSL library.

Changed in version 3.3: `SSLError` used to be a subtype of `socket.error`.

`library`

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as SSL, PEM or X509. The range of possible values depends on the OpenSSL version.

New in version 3.3.

`reason`

A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

New in version 3.3.

`exception ssl.SSLZeroReturnError`
A subclass of `SSLError` raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn’t mean that the underlying transport (read TCP) has been closed.

*New in version 3.3.*

**exception ssl.** `SSLWantReadError`  
A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

*New in version 3.3.*

**exception ssl.** `SSLWantWriteError`  
A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

*New in version 3.3.*

**exception ssl.** `SSLSyscallError`  
A subclass of `SSLError` raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

*New in version 3.3.*

**exception ssl.** `SSLEOFError`  
A subclass of `SSLError` raised when the SSL connection has been terminated abruptly. Generally, you shouldn’t try to reuse the underlying transport when this error is encountered.
New in version 3.3.

```
exception ssl.**SSLCertVerificationError**
A subclass of **SSLError** raised when certificate validation has failed.
```

New in version 3.7.

```
**verify_code**
A numeric error number that denotes the verification error.
```

```
**verify_message**
A human readable string of the verification error.
```

```
exception ssl.**CertificateError**
An alias for **SSLCertVerificationError**.
```

*Changed in version 3.7:* The exception is now an alias for **SSLCertVerificationError**.

### Random generation

```
ssl.**RAND_bytes**(num)
Return *num* cryptographically strong pseudo-random bytes. Raises an **SSLError** if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. **RAND_status()** can be used to check the status of the PRNG and **RAND_add()** can be used to seed the PRNG.
```

For almost all applications **os.urandom()** is preferable.

Read the Wikipedia article, [Cryptographically secure pseudorandom number generator (CSPRNG)](https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator), to get the
requirements of a cryptographically generator.

*New in version 3.3.*

**ssl.** **RAND_pseudo_bytes**(num)

Return (bytes, is_cryptographic): bytes are *num* pseudo-random bytes, is_cryptographic is True if the bytes generated are cryptographically strong. Raises an **SSLError** if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

For almost all applications **os.urandom()** is preferable.

*New in version 3.3.*

*Deprecated since version 3.6:* OpenSSL has deprecated **ssl.RAND_pseudo_bytes()**, use **ssl.RAND_bytes()** instead.

**ssl.** **RAND_status()**

Return True if the SSL pseudo-random number generator has been seeded with ‘enough’ randomness, and False otherwise. You can use **ssl.RAND_egd()** and **ssl.RAND_add()** to increase the randomness of the pseudo-random number generator.

**ssl.** **RAND_egd**(path)

If you are running an entropy-gathering daemon (EGD) somewhere, and *path* is the pathname of a socket connection open
to it, this will read 256 bytes of randomness from the socket, and
add it to the SSL pseudo-random number generator to increase the
security of generated secret keys. This is typically only necessary
on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for
sources of entropy-gathering daemons.

Availability: not available with LibreSSL and OpenSSL > 1.1.0.

ssl. **RAND_add**(bytes, entropy)

Mix the given bytes into the SSL pseudo-random number
generator. The parameter entropy (a float) is a lower bound on the
entropy contained in string (so you can always use 0.0). See RFC
1750 for more information on sources of entropy.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

Certificate handling

ssl. **match_hostname**(cert, hostname)

Verify that cert (in decoded format as returned by
``SSL[['Socket. getpeercert()] matches the given hostname. The
rules applied are those for checking the identity of HTTPS servers
as outlined in RFC 2818, RFC 5280 and RFC 6125. In addition to
HTTPS, this function should be suitable for checking the identity of
servers in various SSL-based protocols such as FTPS, IMAPS,
POPS and others.

CertificateError is raised on failure. On success, the function
returns nothing:

```python
>>> cert = {'subject': ((('commonName', 'example.'
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'

New in version 3.2.

Changed in version 3.3.3: The function now follows RFC 6125, section 6.4.3 and does neither match multiple wildcards (e.g. *.*.com or *a*.example.org) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as www*.xn--pthon-kva.org are still supported, but x*.python.org no longer matches xn--tda.python.org.

Changed in version 3.5: Matching of IP addresses, when present in the subjectAltName field of the certificate, is now supported.

Changed in version 3.7: The function is no longer used to TLS connections. Hostname matching is now performed by OpenSSL.

Allow wildcard when it is the leftmost and the only character in that segment. Partial wildcards like www*.example.com are no longer supported.

Deprecates since version 3.7.

ssl.cert_time_to_seconds(cert_time)
Return the time in seconds since the Epoch, given the cert_time string representing the "notBefore" or "notAfter" date from a certificate in "%b %d %H:%M:%S %Y %Z" strptime format (C locale).
Here's an example:

```python
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan 5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

“notBefore” or “notAfter” dates must use GMT (RFC 5280).

*Changed in version 3.5:* Interpret the input time as a time in UTC as specified by ‘GMT’ timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

```python
ssl.get_server_certificate(addr, ssl_version=PROTOCOL_TLS, ca_certs=None)
```

Given the address `addr` of an SSL-protected server, as a `(hostname, port-number)` pair, fetches the server’s certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in `SSLContext.wrap_socket()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

*Changed in version 3.3:* This function is now IPv6-compatible.

*Changed in version 3.5:* The default `ssl_version` is changed from `PROTOCOL_SSLv3` to `PROTOCOL_TLS` for maximum compatibility.
with modern servers.

ssl.DER_cert_to_PEM_cert(DER_cert_bytes)
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.PEM_cert_to_DER_cert(PEM_cert_string)
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.get_default_verify_paths()
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by SSLContext.set_default_verify_paths(). The return value is a named tuple DefaultVerifyPaths:

- cafile - resolved path to cafile or None if the file doesn't exist,
- capath - resolved path to capath or None if the directory doesn't exist,
- openssl_cafile_env - OpenSSL's environment key that points to a cafile,
- openssl_cafile - hard coded path to a cafile,
- openssl_capath_env - OpenSSL's environment key that points to a capath,
- openssl_capath - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars openssl_cafile_env and openssl_capath_env.

New in version 3.4.

ssl.enum_certificates(store_name)
Retrieve certificates from Windows' system cert store. store_name
may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly True if the certificate is trustworthy for all purposes.

Example:

```python
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

**Availability:** Windows.

*New in version 3.4.*

**ssl.enum_crls(store_name)**

Retrieve CRLs from Windows' system cert store. `store_name` may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data.

**Availability:** Windows.

*New in version 3.4.*
ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version=PROTOCOL_TLS, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)

Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

Internally, function creates a `SSLContext` with protocol `ssl_version` and `SSLContext.options` set to `cert_reqs`. If parameters `keyfile`, `certfile`, `ca_certs` or `ciphers` are set, then the values are passed to `SSLContext.load_cert_chain()`, `SSLContext.load_verify_locations()`, and `SSLContext.set_ciphers()`.

The arguments `server_side`, `do_handshake_on_connect`, and `suppress_ragged_eofs` have the same meaning as `SSLContext.wrap_socket()`.

*Deprecated since version 3.7:* Since Python 3.2 and 2.7.9, it is recommended to use the `SSLContext.wrap_socket()` instead of `wrap_socket()`. The top-level function is limited and creates an insecure client socket without server name indication or hostname matching.

**Constants**

All constants are now `enum.IntEnum` or `enum.IntFlag` collections.
**New in version 3.6.**

**ssl.CERT_NONE**
Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`.
Except for `PROTOCOL_TLS_CLIENT`, it is the default mode.
With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of Security considerations below.

**ssl.CERT_OPTIONAL**
Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`.
In client mode, `CERT_OPTIONAL` has the same meaning as `CERT_REQUIRED`.
It is recommended to use `CERT_REQUIRED` for client-side sockets instead.

In server mode, a client certificate request is sent to the client.
The client may either ignore the request or send a certificate in order perform TLS client cert authentication.
If the client chooses to send a certificate, it is verified.
Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

**ssl.CERT_REQUIRED**
Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode, certificates are required from the other side of the socket connection; an `SSLError` will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. `check_hostname` must be enabled as well to verify the authenticity of a cert. `PROTOCOL_TLS_CLIENT` uses `CERT_REQUIRED` and enables `check_hostname` by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

```python
class ssl.VerifyMode
    enum.IntEnum collection of CERT_* constants.

    New in version 3.6.

ssl.VERIFY_DEFAULT
    Possible value for `SSLContext.verify_flags`. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

    New in version 3.4.

ssl.VERIFY_CRL_CHECK_LEAF
    Possible value for `SSLContext.verify_flags`. In this mode, only the peer cert is check but non of the intermediate CA
certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded `SSLContext.load_verify_locations`, validation will fail.

*New in version 3.4.*

**ssl.VERIFY_CRL_CHECK_CHAIN**
Possible value for `SSLContext.verify_flags`. In this mode, CRLs of all certificates in the peer cert chain are checked.

*New in version 3.4.*

**ssl.VERIFY_X509 STRICT**
Possible value for `SSLContext.verify_flags` to disable workarounds for broken X.509 certificates.

*New in version 3.4.*

**ssl.VERIFY_X509 TRUSTED FIRST**
Possible value for `SSLContext.verify_flags`. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

*New in version 3.4.4.*

**class ssl.VerifyFlags**
`enum.IntFlag` collection of `VERIFY_*` constants.

*New in version 3.6.*

**ssl.PROTOCOL_TLS**
Selects the highest protocol version that both the client and server support. Despite the name, this option can select both “SSL” and
“TLS” protocols.

*New in version 3.6.*

### ssl.**PROTOCOL_TLS_CLIENT**

Auto-negotiate the highest protocol version like `PROTOCOL_TLS`, but only support client-side `SSLSocket` connections. The protocol enables `CERT_REQUIRED` and `check_hostname` by default.

*New in version 3.6.*

### ssl.**PROTOCOL_TLS_SERVER**

Auto-negotiate the highest protocol version like `PROTOCOL_TLS`, but only support server-side `SSLSocket` connections.

*New in version 3.6.*

### ssl.**PROTOCOL_SSLv23**

Alias for data:`PROTOCOL_TLS`.

*Deprecated since version 3.6:* Use `PROTOCOL_TLS` instead.

### ssl.**PROTOCOL_SSLv2**

Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `OPENSSL_NO_SSL2` flag.

**Warning:** SSL version 2 is insecure. Its use is highly discouraged.

*Deprecated since version 3.6:* OpenSSL has removed support for SSLv2.
ssl. **PROTOCOL_SSLv3**

Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the OPENSSL_NO_SSLv3 flag.

**Warning:** SSL version 3 is insecure. Its use is highly discouraged.

*Deprecated since version 3.6:* OpenSSL has deprecated all version specific protocols. Use the default protocol PROTOCOL_TLS with flags like OP_NO_SSLv3 instead.

ssl. **PROTOCOL_TLSv1**

Selects TLS version 1.0 as the channel encryption protocol.

*Deprecated since version 3.6:* OpenSSL has deprecated all version specific protocols. Use the default protocol PROTOCOL_TLS with flags like OP_NO_SSLv3 instead.

ssl. **PROTOCOL_TLSv1_1**

Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

*New in version 3.4.*

*Deprecated since version 3.6:* OpenSSL has deprecated all version specific protocols. Use the default protocol PROTOCOL_TLS with flags like OP_NO_SSLv3 instead.

ssl. **PROTOCOL_TLSv1_2**

Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for
maximum protection, if both sides can speak it. Available only with OpenSSL version 1.0.1+.

*New in version 3.4.*

*Deprecated since version 3.6:* OpenSSL has deprecated all version specific protocols. Use the default protocol `PROTOCOL_TLS` with flags like `OP_NO_SSLv3` instead.

**ssl.OP_ALL**

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL’s `SSL_OP_ALL` constant.

*New in version 3.2.*

**ssl.OP_NO_SSLv2**

Prevents an SSLv2 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv2 as the protocol version.

*New in version 3.2.*

*Deprecated since version 3.6:* SSLv2 is deprecated

**ssl.OP_NO_SSLv3**

Prevents an SSLv3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv3 as the protocol version.

*New in version 3.2.*

*Deprecated since version 3.6:* SSLv3 is deprecated
ssl. **OP_NO_TLSv1**

Prevents a TLSv1 connection. This option is only applicable in conjunction with **PROTOCOL_TLS**. It prevents the peers from choosing TLSv1 as the protocol version.

*New in version 3.2.*

*Deprecated since version 3.7:* The option is deprecated since OpenSSL 1.1.0, use the new `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

ssl. **OP_NO_TLSv1_1**

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with **PROTOCOL_TLS**. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

*New in version 3.4.*

*Deprecated since version 3.7:* The option is deprecated since OpenSSL 1.1.0.

ssl. **OP_NO_TLSv1_2**

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with **PROTOCOL_TLS**. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

*New in version 3.4.*

*Deprecated since version 3.7:* The option is deprecated since OpenSSL 1.1.0.

ssl. **OP_NO_TLSv1_3**
Prevents a TLSv1.3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to 0.

*New in version 3.7.*

*Deprecated since version 3.7:* The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15, 3.6.3 and 3.7.0 for backwards compatibility with OpenSSL 1.0.2.

**`ssl.OP_NO_RENEGOTIATION`**

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

*New in version 3.7.*

**`ssl.OP_CIPHER_SERVER_PREFERENCE`**

Use the server’s cipher ordering preference, rather than the client’s. This option has no effect on client sockets and SSLv2 server sockets.

*New in version 3.3.*

**`ssl.OP_SINGLE_DH_USE`**

Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

*New in version 3.3.*
ssl.**OP_SINGLE_ECDH_USE**
Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

*New in version 3.3.*

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

*New in version 3.8.*

ssl.**OP_NO_COMPRESSION**
Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

*New in version 3.3.*

class ssl.**Options**
`enum.IntFlag` collection of OP_* constants.

ssl.**OP_NO_TICKET**
Prevent client side from requesting a session ticket.

*New in version 3.6.*

ssl.**HAS_ALPN**
Whether the OpenSSL library has built-in support for the
Application-Layer Protocol Negotiation TLS extension as described in RFC 7301.

New in version 3.5.

ssl. HAS_NEVER_CHECK_COMMON_NAME
Whether the OpenSSL library has built-in support not checking subject common name and SSLContext.hostname_checks_common_name is writeable.

New in version 3.7.

ssl. HAS_ECDH
Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

New in version 3.3.

ssl. HAS_SNI
Whether the OpenSSL library has built-in support for the Server Name Indication extension (as defined in RFC 6066).

New in version 3.2.

ssl. HAS_NPN
Whether the OpenSSL library has built-in support for the Next Protocol Negotiation as described in the Application Layer Protocol Negotiation. When true, you can use the SSLContext.set_npn_protocols() method to advertise which protocols you want to support.

New in version 3.3.
ssl.**HAS_SSLv2**  
Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.  

*New in version 3.7.*

ssl.**HAS_SSLv3**  
Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.  

*New in version 3.7.*

ssl.**HAS_TLSv1**  
Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.  

*New in version 3.7.*

ssl.**HAS_TLSv1_1**  
Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.  

*New in version 3.7.*

ssl.**HAS_TLSv1_2**  
Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.  

*New in version 3.7.*

ssl.**HAS_TLSv1_3**  
Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.
New in version 3.7.

**ssl.CHANNEL_BINDING_TYPES**

List of supported TLS channel binding types. Strings in this list can be used as arguments to `SSLSocket.get_channel_binding()`.

New in version 3.3.

**ssl.OPENSSL_VERSION**

The version string of the OpenSSL library loaded by the interpreter:

```python
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k 26 Jan 2017'
```

New in version 3.2.

**ssl.OPENSSL_VERSION_INFO**

A tuple of five integers representing version information about the OpenSSL library:

```python
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

New in version 3.2.

**ssl.OPENSSL_VERSION_NUMBER**

The raw version number of the OpenSSL library, as a single integer:

```python
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```
New in version 3.2.

ssl. ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl. ALERT_DESCRIPTION_INTERNAL_ERROR

ALERT_DESCRIPTION_*

Alert Descriptions from RFC 5246 and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in SSLContext.set_servername_callback().

New in version 3.4.

class ssl.AlertDescription
    enum.IntEnum collection of ALERT_DESCRIPTION_* constants.

New in version 3.6.

Purpose. SERVER_AUTH
Option for create_default_context() and SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

New in version 3.4.

Purpose. CLIENT_AUTH
Option for create_default_context() and SSLContext.load_default_certs(). This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

New in version 3.4.
class ssl.**SSLErrorNumber**

`enum.IntEnum` collection of SSL\_ERROR\_\* constants.

*New in version 3.6.*

class ssl.**TLSVersion**

`enum.IntEnum` collection of SSL and TLS versions for `SSLContext.maximum_version` and `SSLContext.minimum_version`.

*New in version 3.7.*

**TLSVersion.**`MINIMUM_SUPPORTED`

**TLSVersion.**`MAXIMUM_SUPPORTED`

- The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don’t reflect the lowest and highest available TLS/SSL versions.

**TLSVersion.**`SSLv3`

**TLSVersion.**`TLSv1`

**TLSVersion.**`TLSv1_1`

**TLSVersion.**`TLSv1_2`

**TLSVersion.**`TLSv1_3`

- SSL 3.0 to TLS 1.3.
SSL Sockets

```python
class ssl.SSLSocket(socket.socket)
```

SSL sockets provide the following methods of Socket Objects:

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `detach()`
- `fileno()`
- `getpeername()`, `getsockname()`
- `getsockopt()`, `setsockopt()`
- `gettimeout()`, `settimeout()`, `setblocking()`
- `listen()`
- `makefile()`
- `recv()`, `recv_into()` (but passing a non-zero flags argument is not allowed)
- `send()`, `sendall()` (with the same limitation)
- `sendfile()` (but `os.sendfile` will be used for plain-text sockets only, else `send()` will be used)
- `shutdown()`

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method.

*Changed in version 3.5:* The `sendfile()` method was added.
Changed in version 3.5: The shutdown() does not reset the socket timeout each time bytes are received or sent. The socket timeout is now to maximum total duration of the shutdown.

Deprecated since version 3.6: It is deprecated to create a SSLSocket instance directly, use SSLContext.wrap_socket() to wrap a socket.

Changed in version 3.7: SSLSocket instances must to created with wrap_socket(). In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSL sockets also have the following additional methods and attributes:

SSLSocket. **read**(len=1024, buffer=None)

Read up to len bytes of data from the SSL socket and return the result as a bytes instance. If buffer is specified, then read into the buffer instead, and return the number of bytes read.

Raise SSLWantReadError or SSLWantWriteError if the socket is non-blocking and the read would block.

As at any time a re-negotiation is possible, a call to read() can also cause write operations.

Changed in version 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to read up to len bytes.

Deprecated since version 3.6: Use recv() instead of read().

SSLSocket. **write**(buf)
Write *buf* to the SSL socket and return the number of bytes written. The *buf* argument must be an object supporting the buffer interface.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is non-blocking and the write would block.

As at any time a re-negotiation is possible, a call to `write()` can also cause read operations.

*Changed in version 3.5:* The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to write *buf*.

*Deprecated since version 3.6:* Use `send()` instead of `write()`.

---

**Note:** The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

---

`SSLSocket.do_handshake()`  
Perform the SSL setup handshake.

*Changed in version 3.4:* The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the socket's `context` is true.

*Changed in version 3.5:* The socket timeout is no more reset each
time bytes are received or sent. The socket timeout is now to maximum total duration of the handshake.

Changed in version 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function \texttt{match_hostname()} is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is send to the peer.

\texttt{SSLSocket.getpeercert}(\texttt{binary\_form=False})

If there is no certificate for the peer on the other end of the connection, return \texttt{None}. If the SSL handshake hasn't been done yet, raise \texttt{ValueError}.

If the \texttt{binary\_form} parameter is \texttt{False}, and a certificate was received from the peer, this method returns a \texttt{dict} instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them \texttt{subject} (the principal for which the certificate was issued) and \texttt{issuer} (the principal issuing the certificate). If a certificate contains an instance of the \textit{Subject Alternative Name} extension (see \textit{RFC 3280}), there will also be a \texttt{subject\_AltName} key in the dictionary.

The \texttt{subject} and \texttt{issuer} fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

\begin{verbatim}
{ 'issuer': ((('countryName', 'IL'),),
             (('organizationName', 'StartCom Ltd.'),
             (('organizationalUnitName', 'Secure Digital Certificate Signing
\end{verbatim}
Note: To validate a certificate for a particular service, you can use the `match_hostname()` function.

If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket’s role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT_OPTIONAL` or `CERT_REQUIRED`).

`Changed in version 3.2:` The returned dictionary includes additional items such as `issuer` and `notBefore`. 

```python
{"commonName":
 'StartCom Class 2 Primary Intermediate

notAfter': 'Nov 22 08:15:19 2013 GMT',

notBefore': 'Nov 21 03:09:52 2011 GMT',

serialNumber': '95F0',

subject': (((

description', '571208-SLe257oHY9f

countryName', 'US'),),

(stateOrProvinceName', 'California

localityName', 'San Francisco'),)

organizationName', 'Electronic Fr

commonName', '*.eff.org'),),

(emailAddress', 'hostmaster@eff.org or

subjectAltName': ((

DNS', '*.eff.org'), ('DNS',

version': 3})
```
Changed in version 3.4: `ValueError` is raised when the handshake isn’t done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and OCSP URIs.

`SSLSocket.cipher()`  
Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.shared_ciphers()`  
Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. `shared_ciphers()` returns `None` if no connection has been established or the socket is a client socket.

*New in version 3.5.*

`SSLSocket.compression()`  
Return the compression algorithm being used as a string, or `None` if the connection isn’t compressed.

If the higher-level protocol supports its own compression mechanism, you can use `OP_NO_COMPRESSION` to disable SSL-level compression.

*New in version 3.3.*

`SSLSocket.get_channel_binding(cb_type="tls-unique")`  
Get channel binding data for current connection, as a bytes object.
Returns `None` if not connected or the handshake has not been completed.

The `cb_type` parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the `CHANNEL_BINDING_TYPES` list. Currently only the ‘tls-unique’ channel binding, defined by RFC 5929, is supported. `ValueError` will be raised if an unsupported channel binding type is requested.

New in version 3.3.

```python
SSLSocket.selected_alpn_protocol()
```

Return the protocol that was selected during the TLS handshake. If `SSLContext.set_alpn_protocols()` was not called, if the other party does not support ALPN, if this socket does not support any of the client’s proposed protocols, or if the handshake has not happened yet, `None` is returned.

New in version 3.5.

```python
SSLSocket.selected_npn_protocol()
```

Return the higher-level protocol that was selected during the TLS/SSL handshake. If `SSLContext.set_npn_protocols()` was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

New in version 3.3.

```python
SSLSocket.unwrap()
```

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be
used for further communication with the other side of the connection, rather than the original socket.

**SSLSocket**. `verify_client_post_handshake()`

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see `SSLContext.post_handshake_auth`.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn’t met (e.g. not TLS 1.3, PHA not enabled), an `SSLError` is raised.

**Note:** Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises `NotImplementedError`.

*New in version 3.7.1.*

**SSLSocket**. `version()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` is no secure connection is established. As of this writing, possible return values include "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" and "TLSv1.2". Recent OpenSSL versions may define more return values.

*New in version 3.5.*
SSLSocket. **pending**

Returns the number of already decrypted bytes available for read, pending on the connection.

SSLSocket. **context**

The SSLContext object this SSL socket is tied to. If the SSL socket was created using the deprecated `wrap_socket()` function (rather than `SSLContext.wrap_socket()`), this is a custom context object created for this SSL socket.

*New in version 3.2.*

SSLSocket. **server_side**

A boolean which is **True** for server-side sockets and **False** for client-side sockets.

*New in version 3.2.*

SSLSocket. **server_hostname**

Hostname of the server: **str** type, or None for server-side socket or if the hostname was not specified in the constructor.

*New in version 3.2.*

*Changed in version 3.7:* The attribute is now always ASCII text. When **server_hostname** is an internationalized domain name (IDN), this attribute now stores the A-label form ("xn--pythn-mua.org"), rather than the U-label form ("pythön.org").

SSLSocket. **session**

The SSLSession for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before
do_handshake() has been called to reuse a session.

New in version 3.6.

SSLSocket.session_reused

New in version 3.6.
SSL Contexts

*New in version 3.2.*

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

```python
class ssl.SSLContext(protocol=PROTOCOL_TLS)
```

Create a new SSL context. You may pass `protocol` which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified, the default is `PROTOCOL_TLS`; it provides the most compatibility with other versions.

Here’s a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

<table>
<thead>
<tr>
<th>client / server</th>
<th>SSLv2</th>
<th>SSLv3</th>
<th>TLS [3]</th>
<th>TLSv1</th>
<th>TLSv1.1</th>
<th>TLSv1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSLv2</td>
<td>yes</td>
<td>no</td>
<td>no [1]</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SSLv3</td>
<td>no</td>
<td>yes</td>
<td>no [2]</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
### TLS Protocol Support

<table>
<thead>
<tr>
<th>Protocol</th>
<th>TLSv1</th>
<th>TLSv1.1</th>
<th>TLSv1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>support</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>default</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>enabled</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Footnotes

1. SSLContext disables SSLv2 with `OP_NO_SSLv2` by default.
2. SSLContext disables SSLv3 with `OP_NO_SSLv3` by default.
3. TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

**See also:** `create_default_context()` lets the `ssl` module choose security settings for a given purpose.

*Changed in version 3.6:* The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2` (except for `PROTOCOL_SSLv2`), and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers (except for `PROTOCOL_SSLv2`).

### SSLContext

SSLContext objects have the following methods and attributes:

SSLContext.**cert_store_stats(**)
Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```python
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

New in version 3.4.

SSLContext.load_cert_chain(certfile, keyfile=None, password=None)

Load a private key and the corresponding certificate. The certfile string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate’s authenticity. The keyfile string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from certfile as well. See the discussion of Certificates for more information on how the certificate is stored in the certfile.

The password argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the password argument. It will be ignored if the private key is not encrypted and no password is needed.

If the password argument is not specified and a password is
required, OpenSSL’s built-in password prompting mechanism will be used to interactively prompt the user for a password.

An SSLError is raised if the private key doesn’t match with the certificate.

*Changed in version 3.3:* New optional argument `password`.

**SSLContext.load_default_certs** (*purpose=Purpose.SERVER_AUTH*)

Load a set of default “certification authority” (CA) certificates from default locations. On Windows it loads CA certs from the CA and ROOT system stores. On other systems it calls `SSLContext.set_default_verify_paths()`. In the future the method may load CA certificates from other locations, too.

The `purpose` flag specifies what kind of CA certificates are loaded. The default settings `Purpose.SERVER_AUTH` loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). `Purpose.CLIENT_AUTH` loads CA certificates for client certificate verification on the server side.

*New in version 3.4.*

**SSLContext.load_verify_locations** (*cafile=None, capath=None, cadata=None*)

Load a set of “certification authority” (CA) certificates used to validate other peers’ certificates when `verify_mode` is other than `CERT_NONE`. At least one of `cafile` or `capath` must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, `SSLContext.verify_flags` must be configured properly.
The `cafile` string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of `Certificates` for more information about how to arrange the certificates in this file.

The `capath` string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The `cadata` object, if present, is either an ASCII string of one or more PEM-encoded certificates or a bytes-like object of DER-encoded certificates. Like with `capath` extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

*Changed in version 3.4:* New optional argument `cadata`

SSLContext.**get_ca_certs**(binary_form=False)
Get a list of loaded “certification authority” (CA) certificates. If the binary_form parameter is False each list entry is a dict like the output of `SSLSocket.getpeercert()`. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from `capath` unless a certificate was requested and loaded by a SSL connection.

**Note:** Certificates in a capath directory aren’t loaded unless they have been used at least once.

*New in version 3.4.*

SSLContext.**get_ciphers**()
Get a list of enabled ciphers. The list is in order of cipher priority. See `SSLContext.set_ciphers()`.
Example:

```python
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLS
  Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
{'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLS
  Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

On OpenSSL 1.1 and newer the cipher dict contains additional fields:

```python
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLS
  Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
```
Availability: OpenSSL 1.0.2+.

New in version 3.6.

**SSLContext.set_default_verify_paths()**

Load a set of default “certification authority” (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there’s no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

**SSLContext.set_ciphers(ciphers)**

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an SSLError will be raised.

**Note:** when connected, the SSLSocket.cipher() method of
SSL sockets will give the currently selected cipher. OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with `set_ciphers()`.

**SSLContext.set_alpn_protocols(protocols)**

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to [RFC 7301](https://tools.ietf.org/rfc/rfc7301). After a successful handshake, the `SSLSocket.selected_alpn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_ALPN` is False.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise `SSLError` when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, `SSLSocket.selected_alpn_protocol()` returns None.

*New in version 3.5.*

**SSLContext.set_npn_protocols(protocols)**

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation. After a successful handshake, the `SSLSocket.selected_npn_protocol()` method will return the agreed-upon protocol.
This method will raise `NotImplementedError` if `HAS_NPN` is False.

*New in version 3.3.*

**SSLContext.sni_callback**

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in [RFC 6066](https://tools.ietf.org/html/rfc6066) section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If `sni_callback` is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the `ssl.SSLSocket`, the second is a string that represents the server name that the client is intending to communicate (or `None` if the TLS Client Hello does not contain a server name) and the third argument is the original `SSLContext`. The server name argument is text. For internationalized domain name, the server name is an IDN A-label ("xn--pythn-mua.org").

A typical use of this callback is to change the `ssl.SSLSocket`'s `SSLContext.context` attribute to a new object of type `SSLContext` representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like `SSLContext.selected_alpn_protocol()` and
SSLContext.context, SSLSocket.getpeercert(), SSLSocket.getpeercert(), SSLSocket.cipher() and SSLSocket.compress() methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The \texttt{sni\_callback} function must return \texttt{None} to allow the TLS negotiation to continue. If a TLS failure is required, a constant \texttt{ALERT\_DESCRIPTION\_\*} can be returned. Other return values will result in a TLS fatal error with \texttt{ALERT\_DESCRIPTION\_INTERNAL\_ERROR}.

If an exception is raised from the \texttt{sni\_callback} function the TLS connection will terminate with a fatal TLS alert message \texttt{ALERT\_DESCRIPTION\_HANDSHAKE\_FAILURE}.

This method will raise \texttt{NotImplementedError} if the OpenSSL library had \texttt{OPENSSL\_NO\_TLSEXT} defined when it was built.

\textit{New in version 3.7.}

\texttt{SSLContext.set\_servername\_callback(server\_name\_callb)}

This is a legacy API retained for backwards compatibility. When possible, you should use \texttt{sni\_callback} instead. The given \texttt{server\_name\_callback} is similar to \texttt{sni\_callback}, except that when the server hostname is an IDN-encoded internationalized domain name, the \texttt{server\_name\_callback} receives a decoded U-label ("pythön.org").

If there is an decoding error on the server name, the TLS connection will terminate with an
ALERT_DESCRIPTION_INTERNAL_ERROR  fatal  TLS  alert message to the client.

New in version 3.4.

**SSLContext.load_dh_params(dhfile)**

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The `dhfile` parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the **OP_SINGLE_DH_USE** option to further improve security.

New in version 3.3.

**SSLContext.set_ecdh_curve(curve_name)**

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The `curve_name` parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the **OP_SINGLE_ECDH_USE** option to further improve security.

This method is not available if HAS_ECDH is False.

New in version 3.3.

See also:

SSL/TLS & Perfect Forward Secrecy
Wrap an existing Python socket `sock` and return an instance of `SSLContext.sslsocket_class` (default `SSLSocket`). The returned SSL socket is tied to the context, its settings and certificates. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise `SSLError`.

On client connections, the optional parameter `server_hostname` specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying `server_hostname` will raise a `ValueError` if `server_side` is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a
socket.connect(), or whether the application program will call it explicitly, by invoking the `SSLSocket.do_handshake()` method. Calling `SSLSocket.do_handshake()` explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as `True` (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if `False`, it will raise the exceptions back to the caller.

`session`, see `session`.

*Changed in version 3.5:* Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

*Changed in version 3.6:* `session` argument was added.

*Changed in version 3.7:* The method returns on instance of `SSLContext.sslsocket_class` instead of hard-coded `SSLSocket`.

`SSLContext.sslsocket_class`

The return type of `SSLContext.wrap_sockets()`, defaults to `SSLSocket`. The attribute can be overridden on instance of class in order to return a custom subclass of `SSLSocket`.

*New in version 3.7.*

`SSLContext.wrap_bio(incoming, outgoing, server_side=False, server_hostname=None, session=None)`
Wrap the BIO objects *incoming* and *outgoing* and return an instance of attr:`SSLContext.sslobject_class` (default `SSLObject`). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The `server_side`, `server_hostname` and `session` parameters have the same meaning as in `SSLContext.wrap_socket()`.

*Changed in version 3.6*: `session` argument was added.

*Changed in version 3.7*: The method returns an instance of `SSLContext.sslobject_class` instead of hard-coded `SSLObject`.

`SSLContext.sslobject_class`

The return type of `SSLContext.wrap_bio()`, defaults to `SSLObject`. The attribute can be overridden on instance of class in order to return a custom subclass of `SSLObject`.

*New in version 3.7.*

`SSLContext.session_stats()`

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.check_hostname`
Whether to match the peer cert's hostname with `match_hostname()` in `SSLSocket.do_handshake()`. The context's `verify_mode` must be set to `CERT_OPTIONAL` or `CERT_REQUIRED`, and you must pass `server_hostname` to `wrap_socket()` in order to match the hostname. Enabling hostname checking automatically sets `verify_mode` from `CERT_NONE` to `CERT_REQUIRED`. It cannot be set back to `CERT_NONE` as long as hostname checking is enabled.

Example:

```python
import socket, ssl

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com', 443)
ssl_sock.connect(('www.verisign.com', 443))
```

*New in version 3.4.*

*Changed in version 3.7:* `verify_mode` is now automatically changed to `CERT_REQUIRED` when hostname checking is enabled and `verify_mode` is `CERT_NONE`. Previously the same operation would have failed with a `ValueError`.

**Note:** This feature requires OpenSSL 0.9.8f or newer.

`SSLContext.maximum_version`
A `TLSVersion` enum member representing the highest supported TLS version. The value defaults to `TLSVersion.MAXIMUM_SUPPORTED`. The attribute is read-only for protocols other than `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER`.

The attributes `maximum_version`, `minimum_version` and `SSLContext.options` all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with `OP_NO_TLSv1_2` in `options` and `maximum_version` set to `TLSVersion.TLSv1_2` will not be able to establish a TLS 1.2 connection.

**Note:** This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

*New in version 3.7.*

`SSLContext.minimum_version`

Like `SSLContext.maximum_version` except it is the lowest supported version or `TLSVersion.MINIMUM_SUPPORTED`.

**Note:** This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

*New in version 3.7.*

`SSLContext.options`

An integer representing the set of SSL options enabled on this context. The default value is `OP_ALL`, but you can specify other options such as `OP_NO_SSLv2` by ORing them together.
Note: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a `ValueError`.

*Changed in version 3.6:* `SSLContext.options` returns `Options` flags:

```python
>>> ssl.create_default_context().options  # doctest
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

`SSLContext.post_handshake_auth`

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, `SSLContext.verify_mode` must be set to `CERT OPTIONAL` or `CERT_REQUIRED`, too. The actual client cert exchange is delayed until `SSLSocket.verify_client_post_handshake()` is called and some I/O is performed.

**Note:** Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is None and can’t be modified.

*New in version 3.7.1.*
SSLContext.**protocol**

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.**hostname_checks_common_name**

Whether `check_hostname` falls back to verify the cert’s subject common name in the absence of a subject alternative name extension (default: true).

**Note:** Only writeable with OpenSSL 1.1.0 or higher.

*New in version 3.7.*

SSLContext.**verify_flags**

The flags for certificate verification operations. You can set flags like `VERIFY_CRL_CHECK_LEAF` by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

*New in version 3.4.*

*Changed in version 3.6:* SSLContext.verify_flags returns `VerifyFlags` flags:

```python
>>> ssl.create_default_context().verify_flags  #
<VerifyFlags.VERVIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

Whether to try to verify other peers’ certificates and how to behave if verification fails. This attribute must be one of `CERT_NONE`, `CERT_OPTIONAL` or `CERT_REQUIRED`.

*Changed in version 3.6:* SSLContext.verify_mode returns
**VerifyMode** enum:

```python
g>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```
Certificates

Certificates in general are part of a public-key / private-key system. In this system, each principal, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the public key; the other part is kept secret, and is called the private key. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and only with the other part.

A certificate contains information about two principals. It contains the name of a subject, and the subject’s public key. It also contains a statement by a second principal, the issuer, that the subject is who they claim to be, and that this is indeed the subject’s public key. The issuer’s statement is signed with the issuer’s private key, which only the issuer knows. However, anyone can verify the issuer’s statement by finding the issuer’s public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called “notBefore” and “notAfter”.

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as
“PEM” (see RFC 1422), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a certificate chain. This chain should start with the specific certificate for the principal who “is” the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of that certificate, and so on up the chain till you get to a certificate which is self-signed, that is, a certificate which has the same subject and issuer, sometimes called a root certificate. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority’s certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates
If you are going to require validation of the other side of the connection's certificate, you need to provide a “CA certs” file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform’s certificates file can be used by calling `SSLContext.load_default_certs()`, this is done automatically with `create_default_context()`.

**Combined key and certificate**

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` and `wrap_socket()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
---- BEGIN RSA PRIVATE KEY ----
...
 (private key in base64 encoding) ...
---- END RSA PRIVATE KEY ----
---- BEGIN CERTIFICATE ----
...
 (certificate in base64 PEM encoding) ...
---- END CERTIFICATE ----
```

**Self-signed certificates**

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do
this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem
Generating a 1024 bit RSA private key
........+++++
.................................+++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that into your certificate request. 
What you are about to enter **is** what **is** called a Distinguished 
There are quite a few fields but you can leave some 
For some fields there will be a default value, 
If you enter '.', the field will be left blank. 
-----
Country Name (2 letter code) [AU]:US 
State or Province Name (full name) [Some-State]:MyState 
Locality Name (eg, city) []:Some City 
Organization Name (eg, company) [Internet Widgits Pty]:Internet Widgits Pty 
Organizational Unit Name (eg, section) []:My Group 
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization 
Email Address []:ops@myserver.mygroup.myorganization
```

The disadvantage of a self-signed certificate is that it is its own root 
 certificate, and no one else will have it in their cache of known (and 
trusted) root certificates.
Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```python
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```python
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```python
>>> context = ssl.SSLContext()
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")  # this snippet assumes your operating system places a bundle of all CA certificates in /etc/ssl/certs/ca-bundle.crt; if not, you'll get
```
an error and have to adjust the location)

When you use the context to connect to a server, `CERT_REQUIRED` validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```python
>>> conn = context.wrap_socket(socket.socket(socket.
...                      server_hostname="www.python.org")
>>> conn.connect(('www.python.org', 443))
```

You may then fetch the certificate:

```python
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```python
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCert
    http://crl3.digicert.com/sha2-ev-server-g1.crl
    http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'crlDistributionPoints': ('http://ocsp.digicert.com',
    http://cacerts.digicert.com/DigiCert SHA2
    http://crl3.digicert.com/sha2-ev-server-g1.crl
    http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
    (('organizationName', 'DigiCert Inc'),),
    (('organizationalUnitName', 'www.digicert.com'),
    (('commonName', 'DigiCert SHA2 Extended'),
    'notAfter': 'Sep 9 12:00:00 2016 GMT',
    'notBefore': 'Sep 5 00:00:00 2014 GMT',
    'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
    'subject': ((('businessCategory', 'Private Organization'),
    ('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
    (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),
    (('serialNumber', '3359300'),),
    (('streetAddress', '16 Allen Rd'),),
```
Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```python
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
```
See the discussion of Security considerations below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call listen() on it, and start waiting for clients to connect:

```python
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call accept() on the socket to get the new socket from the other end, and use the context's `SSLContext.wrap_socket()` method to create a server-side SSL socket for the connection:
```python
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the connstream and do something with it till you are finished with the client (or the client is finished with you):

```python
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in non-blocking mode and use an event loop).
Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most `SSLSocket` methods will raise either `SSLWantWriteError` or `SSLWantReadError` instead of `BlockingIOError` if an I/O operation would block. `SSLWantReadError` will be raised if a read operation on the underlying socket is necessary, and `SSLWantWriteError` for a write operation on the underlying socket. Note that attempts to write to an SSL socket may require reading from the underlying socket first, and attempts to read from the SSL socket may require a prior write to the underlying socket.

  *Changed in version 3.5:* In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising `SSLWantWriteError` or `SSLWantReadError`.

- Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to `select()`.

- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without `select()` being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and
then only block on a `select()` call if still necessary.

(of course, similar provisions apply when using other primitives such as `poll()`, or those in the `selectors` module)

- The SSL handshake itself will be non-blocking: the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket's readiness:

```python
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [],)
    except ssl.SSLWantWriteError:
        select.select([], [sock], []
```

See also: The `asyncio` module supports non-blocking SSL sockets and provides a higher level API. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.
Memory BIO Support

New in version 3.5.

Ever since the SSL module was introduced in Python 2.6, the SSLSocket class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by socket.socket, from which SSLSocket also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn’t. An example is async IO frameworks that want to use a different IO multiplexing model than the “select/poll on a file descriptor” (readiness based) model that is assumed by socket.socket and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of SSLSocket called SSLObject is provided.

class ssl.SSLObject

A reduced-scope variant of SSLSocket representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object
as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate “BIO” objects which are OpenSSL’s IO abstraction layer.

This class has no public constructor. An SSLObject instance must be created using the wrap_bio() method. This method will create the SSLObject instance and bind it to a pair of BIOs. The incoming BIO is used to pass data from Python to the SSL protocol instance, while the outgoing BIO is used to pass data the other way around.

The following methods are available:

- context
- server_side
- server_hostname
- session
- session_reused
- read()
- write()
- getpeercert()
- selected_npn_protocol()
- cipher()
- shared_ciphers()
- compression()
- pending()
- do_handshake()
- unwrap()
- get_channel_binding()

When compared to SSLSocket, this object lacks the following features:
• Any form of network IO; `recv()` and `send()` read and write only to the underlying `MemoryBIO` buffers.

• There is no `do_handshake_on_connect` machinery. You must always manually call `do_handshake()` to start the handshake.

• There is no handling of `suppress_ragged_eofs`. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.

• The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.

• The `server_name_callback` callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject`:

• All IO on an `SSLObject` is non-blocking. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.

• There is no module-level `wrap_bio()` call like there is for `wrap_socket()`. An `SSLObject` is always created via an `SSLContext`.

*Changed in version 3.7:* `SSLObject` instances must to created with `wrap_bio()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An `SSLObj`ect communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:
class ssl.MemoryBIO

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(n=-1)

Read up to n bytes from the memory buffer. If n is not specified or negative, all bytes are returned.

write(buf)

Write the bytes from buf to the memory BIO. The buf argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of buf.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call write(). The attribute eof will become true after all data currently in the buffer has been read.
SSL session

New in version 3.6.

class ssl.SSLSession
   Session object used by session.

   id
   time
   timeout
   ticket_lifetime_hint
   has_ticket
Security considerations

Best defaults

For **client use**, if you don’t have any special requirements for your security policy, it is highly recommended that you use the `create_default_context()` function to create your SSL context. It will load the system’s trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the `smtplib.SMTP` class to create a trusted, secure connection to a SMTP server:

```python
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with `SSLContext.load_cert_chain()`.

By contrast, if you create the SSL context by calling the `SSLContext` constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates
When calling the `SSLContext` constructor directly, `CERT_NONE` is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use `CERT_REQUIRED`. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling `SSLSocket.getpeercert()`, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the `match_hostname()` function can be used. This common check is automatically performed when `SSLContext.check_hostname` is enabled.

*Changed in version 3.7:* Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify `CERT_REQUIRED` and similarly check the client certificate.

**Protocol versions**

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` as the protocol version. SSLv2 and SSLv3 are disabled by default.

```python
client_context = ssl.SSLContext(ssl.PROTOCOL_TLS
client_context.options |= ssl.OP_NO_TLSv1
client_context.options |= ssl.OP_NO_TLSv1_1
```
The SSL context created above will only allow TLSv1.2 and later (if supported by your system) connections to a server. `PROTOCOL_TLS_CLIENT` implies certificate validation and hostname checks by default. You have to load certificates into the context.

**Cipher selection**

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use `SSLContext.get_ciphers()` or the `openssl ciphers` command on your system.

**Multi-processing**

If using this module as part of a multi-processed application (using, for example the `multiprocessing` or `concurrent.futures` modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with `os.fork()`. Any successful call of `RAND_add()`, `RAND_bytes()` or `RAND_pseudo_bytes()` is sufficient.
TLS 1.3

*New in version 3.7.*

Python has provisional and experimental support for TLS 1.3 with OpenSSL 1.1.1. The new protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method `SSLContext.set_ciphers()` cannot enable or disable any TLS 1.3 ciphers yet, but `SSLContext.get_ciphers()` returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. `SSLSocket.session` and `SSLSession` are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.
LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods SSLContext.set_npn_protocols() and SSLServerSocket.selected_npn_protocol() are not available.
- SSLContext.set_default_verify_paths() ignores the env vars SSL_CERT_FILE and SSL_CERT_PATH although get_default_verify_paths() still reports them.

See also:

Class socket.socket
   Documentation of underlying socket class

SSL/TLS Strong Encryption: An Introduction
   Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
   Steve Kent

RFC 4086: Randomness Requirements for Security
   Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
   D. Cooper

   T. Dierks et. al.
RFC 6066: Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters
IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

Mozilla’s Server Side TLS recommendations
Mozilla
select — Waiting for I/O completion

This module provides access to the select() and poll() functions available in most operating systems, devpoll() available on Solaris and derivatives, epoll() available on Linux 2.5+ and kqueue() available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

**Note:** The selectors module allows high-level and efficient I/O multiplexing, built upon the select module primitives. Users are encouraged to use the selectors module instead, unless they want precise control over the OS-level primitives used.

The module defines the following:

*exception select.error*

A deprecated alias of OSError.

*Changed in version 3.3:* Following PEP 3151, this class was made an alias of OSError.

select.devpoll()

(Only supported on Solaris and derivatives.) Returns a /dev/poll polling object; see section /dev/poll Polling Objects below for the methods supported by devpoll objects.
devpoll() objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, devpoll() will fail. If your program increases this value, devpoll() may return an incomplete list of active file descriptors.

The new file descriptor is **non-inheritable**.

*New in version 3.3.*

*Changed in version 3.4:* The new file descriptor is now non-inheritable.

```
select.epoll(sizehint=-1, flags=0)
```

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

*sizehint* informs epoll about the expected number of events to be registered. It must be positive, or -1 to use the default. It is only used on older systems where epoll_create1() is not available; otherwise it has no effect (though its value is still checked).

*flags* is deprecated and completely ignored. However, when supplied, its value must be 0 or select.EPOLL_CLOEXEC, otherwise OSError is raised.

See the [Edge and Level Trigger Polling (epoll) Objects](#) section below for the methods supported by epolling objects.

epoll objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is **non-inheritable**.
Changed in version 3.3: Added the flags parameter.

Changed in version 3.4: Support for the with statement was added. The new file descriptor is now non-inheritable.

 Deprecated since version 3.4: The flags parameter. select.EPOLL_CLOEXEC is used by default now. Use os.set_inheritable() to make the file descriptor inheritable.

select.poll()
(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section Polling Objects below for the methods supported by polling objects.

select.kqueue()
(Only supported on BSD.) Returns a kernel queue object; see section Kqueue Objects below for the methods supported by kqueue objects.

The new file descriptor is non-inheritable.

Changed in version 3.4: The new file descriptor is now non-inheritable.

select.kevent(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)
(Only supported on BSD.) Returns a kernel event object; see section Kevent Objects below for the methods supported by kevent objects.

select.select(rlist, wlist, xlist[, timeout])
This is a straightforward interface to the Unix select() system
call. The first three arguments are sequences of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- `rlist`: wait until ready for reading
- `wlist`: wait until ready for writing
- `xlist`: wait for an “exceptional condition” (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional `timeout` argument specifies a timeout as a floating point number in seconds. When the `timeout` argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a `wrapper` class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

**Note:** File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.
*Changed in version 3.5:* The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](https://www.python.org/dev/peps/pep-0475/) for the rationale), instead of raising `InterruptedError`.

**select.** `PIPE_BUF`

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by `select()`, `poll()` or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

**Availability:** Unix

*New in version 3.2.*
/dev/poll Polling Objects

Solaris and derivatives have /dev/poll. While select() is O(highest file descriptor) and poll() is O(number of file descriptors), /dev/poll is O(active file descriptors).

/dev/poll behaviour is very close to the standard poll() object.

devpoll.close()
Close the file descriptor of the polling object.

*New in version 3.4.*

devpoll.closed
True if the polling object is closed.

*New in version 3.4.*

devpoll.fileno()
Return the file descriptor number of the polling object.

*New in version 3.4.*

devpoll.register(fd[, eventmask])
Register a file descriptor with the polling object. Future calls to the poll() method will then check whether the file descriptor has any pending I/O events. fd can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

*eventmask* is an optional bitmask describing the type of events you want to check for. The constants are the same that with poll()
object. The default value is a combination of the constants POLLIN, POLLPRI, and POLLOUT.

Warning: Registering a file descriptor that’s already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with poll().

devpoll.modify(fd[, eventmask])

This method does an unregister() followed by a register(). It is (a bit) more efficient than doing the same explicitly.

devpoll.unregister(fd)

Remove a file descriptor being tracked by a polling object. Just like the register() method, fd can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

devpoll.poll([timeout])

Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. fd is the file descriptor, and event is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If timeout is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If timeout is omitted, -1, or None, the call will block until there is an
event for this poll object.

*Changed in version 3.5:* The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see PEP 475 for the rationale), instead of raising `InterruptedError`. 
Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPOLLIN</td>
<td>Available for read</td>
</tr>
<tr>
<td>EPOLLOUT</td>
<td>Available for write</td>
</tr>
<tr>
<td>EPOLLPRI</td>
<td>Urgent data for read</td>
</tr>
<tr>
<td>EPOLLERR</td>
<td>Error condition happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLHUP</td>
<td>Hang up happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLET</td>
<td>Set Edge Trigger behavior, the default is Level Trigger behavior</td>
</tr>
<tr>
<td>EPOLLONESSHOT</td>
<td>Set one-shot behavior. After one event is pulled out, the fd is internally disabled</td>
</tr>
<tr>
<td>EPOLLEXCLUSIVE</td>
<td>Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.</td>
</tr>
<tr>
<td>EPOLLRDHUP</td>
<td>Stream socket peer closed connection or shut down writing half of connection.</td>
</tr>
<tr>
<td>EPOLLRDNORM</td>
<td>Equivalent to EPOLLIN</td>
</tr>
<tr>
<td>EPOLLRDBAND</td>
<td>Priority data band can be read</td>
</tr>
<tr>
<td>EPOLLWRNORM</td>
<td>Equivalent to EPOLLOUT</td>
</tr>
<tr>
<td>EPOLLWRBAND</td>
<td>Priority data may be written</td>
</tr>
<tr>
<td>EPOLLMSG</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

*New in version 3.6:* EPOLLEXCLUSIVE was added. It’s only supported by Linux Kernel 4.5 or later.

```python
epoll.close()
```

Close the control file descriptor of the epoll object.

```python
epoll.closed
```

```
True
```

if the epoll object is closed.

```python
epoll.fileno()
```

Return the file descriptor number of the control fd.

```python
epoll.fromfd(fd)
```

Create an epoll object from a given file descriptor.

```python
epoll.register(fd[, eventmask])
```

Register a fd descriptor with the epoll object.

```python
epoll.modify(fd, eventmask)
```

Modify a registered file descriptor.

```python
epoll.unregister(fd)
```

Remove a registered file descriptor from the epoll object.

```python
epoll.poll(timeout=-1, maxevents=-1)
```

Wait for events. timeout in seconds (float)

*Changed in version 3.5:* The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see PEP 475 for the rationale), instead of raising `InterruptedError`. 
Polling Objects

The poll() system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. poll() scales better because the system call only requires listing the file descriptors of interest, while select() builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. select() is $O$(highest file descriptor), while poll() is $O$(number of file descriptors).

`poll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the poll() method will then check whether the file descriptor has any pending I/O events. $fd$ can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

`eventmask` is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants POLLIN, POLLPRI, and POLLOUT, described in the table below. If not specified, the default value used will check for all 3 types of events.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>There is data to read</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>There is urgent data to read</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Ready for output: writing will not block</td>
</tr>
<tr>
<td>POLLERR</td>
<td>Error condition of some sort</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>Hung up</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>POLLRDHUP</td>
<td>Stream socket peer closed connection, or shut down writing half of connection</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>Invalid request: descriptor not open</td>
</tr>
</tbody>
</table>

Registering a file descriptor that’s already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.modify(fd, eventmask)`

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an `OSError` exception with errno `ENOENT` to be raised.

`poll.unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, `fd` can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a `KeyError` exception to be raised.

`poll.poll([timeout])`

Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. `fd` is the file descriptor, and `event` is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If `timeout` is given, it specifies the length of time in milliseconds which the system will wait for events before returning.
If `timeout` is omitted, negative, or `None`, the call will block until there is an event for this poll object.

*Changed in version 3.5:* The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](https://www.python.org/dev/peps/pep-0475) for the rationale), instead of raising `InterruptedError`.
Kqueue Objects

kqueue.close()
    Close the control file descriptor of the kqueue object.

kqueue.closed
    True if the kqueue object is closed.

kqueue.fileno()
    Return the file descriptor number of the control fd.

kqueue.fromfd(fd)
    Create a kqueue object from a given file descriptor.

kqueue.control(changelist, max_events[, timeout=None]) → eventlist
    Low level interface to kevent

    • changelist must be an iterable of kevent object or None
    • max_events must be 0 or a positive integer
    • timeout in seconds (floats possible)

*Changed in version 3.5:* The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see PEP 475 for the rationale), instead of raising InterruptedError.
Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent. **ident**

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a `fileno()` method. kevent stores the integer internally.

kevent. **filter**

Name of the kernel filter.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_FILTER_READ</td>
<td>Takes a descriptor and returns whenever there is data available to read</td>
</tr>
<tr>
<td>KQ_FILTER_WRITE</td>
<td>Takes a descriptor and returns whenever there is data available to write</td>
</tr>
<tr>
<td>KQ_FILTER_AIO</td>
<td>AIO requests</td>
</tr>
<tr>
<td>KQ_FILTER_VNODE</td>
<td>Returns when one or more of the requested events watched in flag occurs</td>
</tr>
<tr>
<td>KQ_FILTER_PROC</td>
<td>Watch for events on a process id</td>
</tr>
<tr>
<td>KQ_FILTER_NETDEV</td>
<td>Watch for events on a network device [not available on Mac OS X]</td>
</tr>
<tr>
<td>KQ_FILTER_SIGNAL</td>
<td>Returns whenever the watched signal is delivered to the process</td>
</tr>
<tr>
<td>KQ_FILTER_TIMER</td>
<td>Establishes an arbitrary timer</td>
</tr>
</tbody>
</table>
**kevent. flags**
Filter action.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_EV_ADD</td>
<td>Adds or modifies an event</td>
</tr>
<tr>
<td>KQ_EV_DELETE</td>
<td>Removes an event from the queue</td>
</tr>
<tr>
<td>KQ_EV_ENABLE</td>
<td>Permits control() to return the event</td>
</tr>
<tr>
<td>KQ_EV_DISABLE</td>
<td>Disable event</td>
</tr>
<tr>
<td>KQ_EV_ONESHOT</td>
<td>Removes event after first occurrence</td>
</tr>
<tr>
<td>KQ_EV_CLEAR</td>
<td>Reset the state after an event is retrieved</td>
</tr>
<tr>
<td>KQ_EV_SYSFLAGS</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_FLAG1</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_EOF</td>
<td>Filter specific EOF condition</td>
</tr>
<tr>
<td>KQ_EV_ERROR</td>
<td>See return values</td>
</tr>
</tbody>
</table>

**kevent. flags**
Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_LOWAT</td>
<td>low water mark of a socket buffer</td>
</tr>
</tbody>
</table>

KQ_FILTER_VNODE filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_DELETE</td>
<td><code>unlink()</code> was called</td>
</tr>
<tr>
<td>Constant</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>KQ_NOTE_WRITE</td>
<td>a write occurred</td>
</tr>
<tr>
<td>KQ_NOTE_EXTEND</td>
<td>the file was extended</td>
</tr>
<tr>
<td>KQ_NOTE_ATTRIB</td>
<td>an attribute was changed</td>
</tr>
<tr>
<td>KQ_NOTE_LINK</td>
<td>the link count has changed</td>
</tr>
<tr>
<td>KQ_NOTE_RENAME</td>
<td>the file was renamed</td>
</tr>
<tr>
<td>KQ_NOTE_REVOKE</td>
<td>access to the file was revoked</td>
</tr>
</tbody>
</table>

**KQ_FILTER_PROC filter flags:**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_EXIT</td>
<td>the process has exited</td>
</tr>
<tr>
<td>KQ_NOTE_FORK</td>
<td>the process has called <code>fork()</code></td>
</tr>
<tr>
<td>KQ_NOTE_EXEC</td>
<td>the process has executed a new process</td>
</tr>
<tr>
<td>KQ_NOTE_PCTRLMASK</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>KQ_NOTE_PDATAMASK</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>KQ_NOTE_TRACK</td>
<td>follow a process across <code>fork()</code></td>
</tr>
<tr>
<td>KQ_NOTE_CHILD</td>
<td>returned on the child process for <code>NOTE_TRACK</code></td>
</tr>
<tr>
<td>KQ_NOTE_TRACKERR</td>
<td>unable to attach to a child</td>
</tr>
</tbody>
</table>

**KQ_FILTER_NETDEV filter flags (not available on Mac OS X):**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_NOTE_LINKUP</td>
<td>link is up</td>
</tr>
<tr>
<td>KQ_NOTE_LINKDOWN</td>
<td>link is down</td>
</tr>
<tr>
<td>KQ_NOTE_LINKINV</td>
<td>link state is invalid</td>
</tr>
</tbody>
</table>
kevent.data
Filter specific data.

kevent.udata
User defined value.
selectors — High-level I/O multiplexing

New in version 3.4.

Source code: Lib/selectors.py
Introduction

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, “file object” refers to any object with a `fileno()` method, or a raw file descriptor. See `file object`.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

**Note:** The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

**See also:**

`select`

Low-level I/O multiplexing module.
Classes

Classes hierarchy:

```
BaseSelector
  +- SelectSelector
  +- PollSelector
  +- EpollSelector
  +- DevpollSelector
  ++ KqueueSelector
```

In the following, `events` is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_READ</td>
<td>Available for read</td>
</tr>
<tr>
<td>EVENT_WRITE</td>
<td>Available for write</td>
</tr>
</tbody>
</table>

*class selectors.**SelectorKey**

A **SelectorKey** is a **namedtuple** used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several **BaseSelector** methods.

**fileobj**

File object registered.

**fd**

Underlying file descriptor.

**events**

Events that must be waited for on this file object.
**data**

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

**class selectors. BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It’s an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector, KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the [context manager](https://docs.python.org/3/glossary.html#context-manager) protocol.

*abstractmethod register*(fileobj, events, data=None)

Register a file object for selection, monitoring it for I/O events.

*fileobj* is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

*abstractmethod unregister*(fileobj)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

*fileobj* must be a file object previously registered.
This returns the associated `SelectorKey` instance, or raises a `KeyError` if `fileobj` is not registered. It will raise a `ValueError` if `fileobj` is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

**modify(fileobj, events, data=None)**

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)()` followed by `BaseSelector.register(fileobj, events, data)()`, except that it can be implemented more efficiently.

This returns a new `SelectorKey` instance, or raises a `ValueError` in case of invalid event mask or file descriptor, or `KeyError` if the file object is not registered.

**abstractmethod select(timeout=None)**

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If `timeout` is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

`key` is the `SelectorKey` instance corresponding to a ready file object. `events` is a bitmask of events ready on this file object.
Note: This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

Changed in version 3.5: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see PEP 475 for the rationale), instead of returning an empty list of events before the timeout.

close()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(fileobj)

Return the key associated with a registered file object.

This returns the SelectorKey instance associated to this file object, or raises KeyError if the file object is not registered.

abstractmethod get_map()

Return a mapping of file objects to selector keys.

This returns a Mapping instance mapping registered file objects to their associated SelectorKey instance.

class selectors.DefaultSelector

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.SelectSelector
select.select() -based selector.

class selectors.PollSelector
select.poll() -based selector.

class selectors.EpollSelector
select.epoll() -based selector.

fileno()
This returns the file descriptor used by the underlying select.epoll() object.

class selectors.DevpollSelector
select.devpoll() -based selector.

fileno()
This returns the file descriptor used by the underlying select.devpoll() object.

New in version 3.5.

class selectors.KqueueSelector
select.kqueue() -based selector.

fileno()
This returns the file descriptor used by the underlying select.kqueue() object.
Examples

Here is a simple echo server implementation:

```python
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
```
callback(key.fileobj, mask)
asyncore — Asynchronous socket handler

Source code: Lib/asyncore.py

Deprecated since version 3.6: Please use asyncio instead.

**Note:** This module exists for backwards compatibility only. For new code we recommend using asyncio.

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do “more than one thing at a time.” Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It’s really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the select() system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the “background.” Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The asyncore module solves many of the difficult problems for you,
making the task of building sophisticated high-performance network servers and clients a snap. For “conversational” applications and protocols the companion `asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network channels, instances of class `asyncore.dispatcher` and `asynchat.async_chat`. Creating the channels adds them to a global map, used by the `loop()` function if you do not provide it with your own `map`.

Once the initial channel(s) is(are) created, calling the `loop()` function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

```python
asyncore.loop([timeout[, use_poll[, map[, count]]]])
```

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The `count` parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The `timeout` argument sets the timeout parameter for the appropriate `select()` or `poll()` call, measured in seconds; the default is 30 seconds. The `use_poll` parameter, if true, indicates that `poll()` should be used in preference to `select()` (the default is `False`).

The `map` parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If `map` is omitted, a global map is used. Channels (instances of `asyncore.dispatcher`, `asynchat.async_chat` and subclasses thereof) can freely be mixed in the map.

```python
class asyncore.dispatcher
```
The dispatcher class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_connect()</td>
<td>Implied by the first read or write event</td>
</tr>
<tr>
<td>handle_close()</td>
<td>Implied by a read event with no data available</td>
</tr>
<tr>
<td>handle_accepted()</td>
<td>Implied by a read event on a listening socket</td>
</tr>
</tbody>
</table>

During asynchronous processing, each mapped channel's readable() and writable() methods are used to determine whether the channel's socket should be added to the list of channels select()ed or poll()ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

**handle_read()**

Called when the asynchronous loop detects that a read() call
on the channel's socket will succeed.

**handle_write()**
Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```python
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

**handle_expt()**
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

**handle_connect()**
Called when the active opener’s socket actually makes a connection. Might send a “welcome” banner, or initiate a protocol negotiation with the remote endpoint, for example.

**handle_close()**
Called when the socket is closed.

**handle_error()**
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

**handle_accept()**
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a `connect()` call for the local endpoint. Deprecated in version 3.2; use `handle_accepted()` instead.
Deprecated since version 3.2.

**handle_accepted**(sock, addr)
Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a `connect()` call for the local endpoint. `sock` is a new socket object usable to send and receive data on the connection, and `addr` is the address bound to the socket on the other end of the connection.

*New in version 3.2.*

**readable()**
Called each time around the asynchronous loop to determine whether a channel’s socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

**writable()**
Called each time around the asynchronous loop to determine whether a channel’s socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

**create_socket**(family=socket.AF_INET,
  type=socket.SOCK_STREAM)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the `socket`
documentation for information on creating sockets.

*Changed in version 3.3: family and type arguments can be omitted.*

**connect(address)**
As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

**send(data)**
Send *data* to the remote end-point of the socket.

**recv(buffer_size)**
Read at most *buffer_size* bytes from the socket's remote end-point. An empty bytes object implies that the channel has been closed from the other end.

Note that *recv()* may raise *BlockingIOError*, even though *select.select()* or *select.poll()* has reported the socket ready for reading.

**listen(backlog)**
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**bind(address)**
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — refer to the socket documentation for more information.) To mark the socket as re-usable (setting the SO_REUSEADDR
option), call the dispatcher object's set_reuse_addr() method.

**accept()**
Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either None or a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection. When None is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

**close()**
Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

**class asyncore.dispatcher_with_send**
A dispatcher subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use asynchat.async_chat.

**class asyncore.file_dispatcher**
A file_dispatcher takes a file descriptor or file object along with an optional map argument and wraps it for use with the poll() or loop() functions. If provided a file object or anything with a fileno() method, that method will be called and passed to the file_wrapper constructor.
Availability: Unix.

class asyncore.file_wrapper
A file_wrapper takes an integer file descriptor and calls os.dup() to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the file_dispatcher class.

Availability: Unix.
Here is a very basic HTTP client that uses the `dispatcher` class to implement its socket handling:

```python
import asyncore

class HTTPClient(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect((host, 80))
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\n' % (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

client = HTTPClient('www.python.org', '/')
asyncore.loop()
```
asyncore Example basic echo server

Here is a basic echo server that uses the dispatcher class to accept connections and dispatches the incoming connections to a handler:

```python
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):
    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind(((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```
asynchat — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

Deprecated since version 3.6: Please use asyncio instead.

Note: This module exists for backwards compatibility only. For new code we recommend using asyncio.

This module builds on the asyncore infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. asynchat defines the abstract class async_chat that you subclass, providing implementations of the collect_incoming_data() and found_terminator() methods. It uses the same asynchronous loop as asyncore, and the two types of channel, asyncore.dispatcher and asynchat.async_chat, can freely be mixed in the channel map. Typically an asyncore.dispatcher server channel generates new asynchat.async_chat channel objects as it receives incoming connection requests.

class asynchat.async_chat

This class is an abstract subclass of asyncore.dispatcher. To make practical use of the code you must subclass async_chat, providing meaningful collect_incoming_data() and found_terminator() methods. The asyncore.dispatcher methods can be used, although not all make sense in a
Like `asyncore.dispatcher`, `async_chat` defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the `async_chat` object’s methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

**ac_in_buffer_size**
- The asynchronous input buffer size (default 4096).

**ac_out_buffer_size**
- The asynchronous output buffer size (default 4096).

Unlike `asyncore.dispatcher`, `async_chat` allows you to define a FIFO queue of producers. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (i.e. that it contains no more data) by having its `more()` method return the empty bytes object. At this point the `async_chat` object removes the producer from the queue and starts using the next producer, if any. When the producer queue is empty the handle_write() method does nothing. You use the channel object’s `set_terminator()` method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning `async_chat` subclass your input methods `collect_incoming_data()` and `found_terminator()` must handle the data that the channel receives asynchronously. The
methods are described below.

**async_chat.close_when_done()**
Pushes a `None` on to the producer queue. When this producer is popped off the queue it causes the channel to be closed.

**async_chat.collect_incoming_data(data)**
Called with `data` holding an arbitrary amount of received data. The default method, which must be overridden, raises a `NotImplementedError` exception.

**async_chat.discard_buffers()**
In emergencies this method will discard any data held in the input and/or output buffers and the producer queue.

**async_chat.found_terminator()**
Called when the incoming data stream matches the termination condition set by `set_terminator()`. The default method, which must be overridden, raises a `NotImplementedError` exception. The buffered input data should be available via an instance attribute.

**async_chat.get_terminator()**
Returns the current terminator for the channel.

**async_chat.push(data)**
Pushes data on to the channel's queue to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

**async_chat.push_with_producer(producer)**
Takes a producer object and adds it to the producer queue associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

```python
async_chat.set_terminator(term)
```

Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

<table>
<thead>
<tr>
<th><code>term</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>Will call <code>found_terminator()</code> when the string is found in the input stream</td>
</tr>
<tr>
<td><code>integer</code></td>
<td>Will call <code>found_terminator()</code> when the indicated number of characters have been received</td>
</tr>
<tr>
<td><code>None</code></td>
<td>The channel continues to collect data forever</td>
</tr>
</tbody>
</table>

Note that any data following the terminator will be available for reading by the channel after `found_terminator()` is called.
asynchat Example

The following partial example shows how HTTP requests can be read with `async_chat`. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```python
import asynchat
class http_request_handler(asynchat.async_chat):
    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b"
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log
```
```python
def collect_incoming_data(self, data):
    """Buffer the data""
    self.ibuffer.append(data)

def found_terminator(self):
    if self.reading_headers:
        self.reading_headers = False
        self.parse_headers(b""".join(self.ibuffer)
        self.ibuffer = []
        if self.op.upper() == b"POST":
            clen = self.headers.getheader("content-length")
            self.set_terminator(int(clen))
        else:
            self.handling = True
            self.set_terminator(None)
            self.handle_request()
    elif not self.handling:
        self.set_terminator(None)  # browsers sometimes over-send
        self.cgi_data = parse(self.headers, b""
        self.handling = True
        self.ibuffer = []
        self.handle_request()
```

signal — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.
General rules

The `signal.signal()` function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a `KeyboardInterrupt` exception.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the virtual machine to execute the corresponding Python signal handler at a later point (for example at the next bytecode instruction). This has consequences:

- It makes little sense to catch synchronous errors like SIGFPE or SIGSEGV that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when
the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can’t be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread is allowed to set a new signal handler.
Module contents

Changed in version 3.5: signal (SIG*), handler (SIG_DFL, SIG_IGN) and sigmask (SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK) related constants listed below were turned into enums. getsignal(), pthread_sigmask(), sigpending() and sigwait() functions return human-readable enums.

The variables defined in the signal module are:

signal.SIG_DFL
This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCHLD is to simply ignore it.

signal.SIG_IGN
This is another standard signal handler, which will simply ignore the given signal.

SIG*
All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for ‘signal()’ lists the existing signals (on some systems this is signal(2), on others the list is in signal(7)). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.CTRL_C_EVENT
The signal corresponding to the `Ctrl+C` keystroke event. This signal can only be used with `os.kill()`.

**Availability:** Windows.

*New in version 3.2.*

**signal.CTRL_BREAK_EVENT**

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with `os.kill()`.

**Availability:** Windows.

*New in version 3.2.*

**signal.NSIG**

One more than the number of the highest signal number.

**signal.ITIMER_REAL**

Decrements interval timer in real time, and delivers SIGALRM upon expiration.

**signal.ITIMER_VIRTUAL**

Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

**signal.ITIMER_PROF**

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

**signal.SIG_BLOCK**
A possible value for the `how` parameter to `pthread_sigmask()` indicating that signals are to be blocked.

*New in version 3.3.*

**signal**. `SIG_UNBLOCK`  
A possible value for the `how` parameter to `pthread_sigmask()` indicating that signals are to be unblocked.

*New in version 3.3.*

**signal**. `SIG_SETMASK`  
A possible value for the `how` parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

*New in version 3.3.*

The `signal` module defines one exception:

**exception** `signal`. `ItimerError`  
Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `OSError`.

*New in version 3.3:* This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

The `signal` module defines the following functions:

**signal**. `alarm(time)`  
If `time` is non-zero, this function requests that a SIGALRM signal be sent to the process in `time` seconds. Any previously scheduled
alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If time is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page alarm(2).)

**Availability:** Unix.

**signal.getsignal(signum)**

Return the current signal handler for the signal *signum*. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

**signal.pause()**

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page signal(2).)

See also `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `sigpending()`.

**signal.pthread_kill(thread_id, signalnum)**

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be executed by the main thread. Therefore, the only point of sending a
signal to a particular Python thread would be to force a running system call to fail with `InterruptedError`.

Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for `thread_id`.

If `signalnum` is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

**Availability:** Unix (see the man page `pthread_kill(3)` for further information).

See also `os.kill()`.

*New in version 3.3.*

`signal.pthread_sigmask`(how, mask)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of `how`, as follows:

- **SIG_BLOCK**: The set of blocked signals is the union of the current set and the `mask` argument.
- **SIG_UNBLOCK**: The signals in `mask` are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- **SIG_SETMASK**: The set of blocked signals is set to the `mask` argument.

`mask` is a set of signal numbers (e.g. `{signal.SIGINT,
signal.SIGTERM). Use `range(1, signal.NSIG)` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

**Availability:** Unix. See the man page `sigprocmask(3)` and `pthread_sigmask(3)` for further information.

See also `pause()`, `sigpending()` and `sigwait()`.

*New in version 3.3.*

`signal.setitimer(which, seconds, interval=0.0)`

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by `which` to fire after `seconds` (float is accepted, different from `alarm()`)) and after that every `interval` seconds (if `interval` is non-zero). The interval timer specified by `which` can be cleared by setting `seconds` to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; `signal.ITIMER_REAL` will deliver SIGALRM, `signal.ITIMER_VIRTUAL` sends SIGVTALRM, and `signal.ITIMER_PROF` will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an `ItimerError`.
Availability: Unix.

**signal.getitimer**(which)

Returns current value of a given interval timer specified by `which`.

Availability: Unix.

**signal.set_wakeup_fd**(fd, *, warn_on_full_buffer=True)

Set the wakeup file descriptor to `fd`. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If `fd` is -1, file descriptor wakeup is disabled. If not -1, `fd` must be non-blocking. It is up to the library to remove any bytes from `fd` before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine `which` signal or signals have arrived.

In the first approach, we read the data out of the fd’s buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set
warn_on_full_buffer=True, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd only for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd’s buffer is empty or non-empty; a full buffer doesn’t indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

*Changed in version 3.5:* On Windows, the function now also supports socket handles.

*Changed in version 3.7:* Added `warn_on_full_buffer` parameter.

```python
signal.siginterrupt(signalnum, flag)
```

Change system call restart behaviour: if `flag` is `False`, system calls will be restarted when interrupted by signal `signalnum`, otherwise system calls will be interrupted. Returns nothing.

**Availability:** Unix (see the man page `siginterrupt(3)` for further information).

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true `flag` value for the given signal.

```python
signal.signal(signalnum, handler)
```

Set the handler for signal `signalnum` to the function `handler`. `handler` can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned
(see the description of `getsignal()` above). (See the Unix man page `signal(2)`.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The `handler` is called with two arguments: the signal number and the current stack frame (None or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, or SIGBREAK. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level constant.

**signal. `sigpending()`**

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

**Availability:** Unix (see the man page `sigpending(2)` for further information).

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

*New in version 3.3.*

**signal. `sigwait(sigset)`**

Suspend execution of the calling thread until the delivery of one of
the signals specified in the signal set *sigset*. The function accepts
the signal (removes it from the pending list of signals), and returns
the signal number.

**Availability:** Unix (see the man page *sigwait(3)* for further
information).

See also *pause(), pthread_sigmask(), sigpending(),
sigwaitinfo() and sigtimedwait()*.

*New in version 3.3.*

**signal.** *sigwaitinfo(sigset)*

Suspend execution of the calling thread until the delivery of one of
the signals specified in the signal set *sigset*. The function accepts
the signal and removes it from the pending list of signals. If one of
the signals in *sigset* is already pending for the calling thread, the
function will return immediately with information about that signal.
The signal handler is not called for the delivered signal. The
function raises an *InterruptedError* if it is interrupted by a
signal that is not in *sigset*.

The return value is an object representing the data contained in the
*siginfo_t* structure, namely: *si_signo, si_code, si_errno,
si_pid, si_uid, si_status, si_band*. 

**Availability:** Unix (see the man page *sigwaitinfo(2)* for further
information).

See also *pause(), sigwait() and sigtimedwait()*.

*New in version 3.3.*

*Changed in version 3.5:* The function is now retried if interrupted by
a signal not in *sigset* and the signal handler does not raise an exception (see PEP 475 for the rationale).

**signal.sigtimedwait(sigset, timeout)**

Like *sigwaitinfo()* , but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as 0, a poll is performed. Returns *None* if a timeout occurs.

**Availability:** Unix (see the man page *sigtimedwait(2)* for further information).

See also *pause()* , *sigwait()* and *sigwaitinfo()* .

*New in version 3.3.*

*Changed in version 3.5:* The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see PEP 475 for the rationale).
Example

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```python
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)  # Disable the alarm
```
Note on SIGPIPE

Piping output of your program to tools like `head(1)` will cause a SIGPIPE signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```python
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output to devnull to avoid another BrokenPipeError.
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code

if __name__ == '__main__':
    main()
```

Do not set SIGPIPE's disposition to `SIG_DFL` in order to avoid `BrokenPipeError`. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.
mmap — Memory-mapped file support

Memory-mapped file objects behave like both bytearray and like file objects. You can use mmap objects in most places where bytearray are expected; for example, you can use the re module to search through a memory-mapped file. You can also change a single byte by doing obj[index] = 97, or change a subsequence by assigning to a slice: obj[i1:i2] = b'...'. You can also read and write data starting at the current file position, and seek() through the file to different positions.

A memory-mapped file is created by the mmap constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its fileno() method to obtain the correct value for the fileno parameter. Otherwise, you can open the file using the os.open() function, which returns a file descriptor directly (the file still needs to be closed when done).

**Note:** If you want to create a memory-mapping for a writable, buffered file, you should flush() the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, access may be specified as an optional keyword parameter. access accepts one of four values: ACCESS_READ, ACCESS_WRITE, or ACCESS_COPY to specify read-only, write-through or copy-on-write.
memory respectively, or ACCESS_DEFAULT to defer to prot. access can be used on both Unix and Windows. If access is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an ACCESS_READ memory map raises a TypeError exception. Assignment to an ACCESS_WRITE memory map affects both memory and the underlying file. Assignment to an ACCESS_COPY memory map affects memory but does not update the underlying file.

*Changed in version 3.7:* Added ACCESS_DEFAULT constant.

To map anonymous memory, -1 should be passed as the fileno along with the length.

```python
class mmap
    mmap(fileno, length, tagname=None, access=ACCESS_DEFAULT[, offset])
```

(Windows version) Maps length bytes from the file specified by the file handle fileno, and creates a mmap object. If length is larger than the current size of the file, the file is extended to contain length bytes. If length is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.
offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of the ALLOCATIONGRANULARITY.

class mmap

```
    mmap(fileno, length, flags=MAP_SHARED,
         prot=PROT_WRITE|PROT_READ, access=ACCESS_DEFAULT[, offset])
```

(Unix version) Maps length bytes from the file specified by the file descriptor fileno, and returns a mmap object. If length is 0, the maximum length of the map will be the current size of the file when mmap is called.

flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that’s shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. prot defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of flags and prot as an optional keyword parameter. It is an error to specify both flags, prot and access. See the description of access above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of
ALLOCATION GRANULARITY which is equal to PAGESIZE on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor `fileno` is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using `mmap`:

```python
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello world"
    # close the map
    mm.close()
```

`mmap` can also be used as a context manager in a `with` statement:

```python
import mmap
```
with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")

New in version 3.2: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```python
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")
pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())
    mm.close()
```

Memory-mapped file objects support the following methods:

**close()**
Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

**closed**
True if the file is closed.

New in version 3.2.
**find**(*sub[, start[, end]]*)

Returns the lowest index in the object where the subsequence `sub` is found, such that `sub` is contained in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Returns `-1` on failure.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

**flush**(*[offset[, size]]*)

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If `offset` and `size` are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. `offset` must be a multiple of the PAGESIZE or ALLOCATIONGRANULARITY.

*(Windows version)* A nonzero value returned indicates success; zero indicates failure.

*(Unix version)* A zero value is returned to indicate success. An exception is raised when the call failed.

**move**(*dest, src, count*)

Copy the `count` bytes starting at offset `src` to the destination index `dest`. If the mmap was created with ACCESS_READ, then calls to move will raise a `TypeError` exception.

**read**(*[, n]*)

Return a bytes containing up to `n` bytes starting from the current file position. If the argument is omitted, `None` or
negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

*Changed in version 3.3:* Argument can be omitted or `None`.

**read_byte()**

Returns a byte at the current file position as an integer, and advances the file position by 1.

**readline()**

Returns a single line, starting at the current file position and up to the next newline.

**resize(newsize)**

Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or `ACCESS_COPY`, resizing the map will raise a `TypeError` exception.

**rfind(sub[, start[, end]])**

Returns the highest index in the object where the subsequence `sub` is found, such that `sub` is contained in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Returns `-1` on failure.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

**seek(pos[, whence])**

Set the file’s current position. `whence` argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current
position) and os.SEEK_END or 2 (seek relative to the file's end).

**size()**
Return the length of the file, which can be larger than the size of the memory-mapped area.

**tell()**
Returns the current position of the file pointer.

**write(bytes)**
Write the bytes in bytes into memory at the current position of the file pointer and return the number of bytes written (never less than len(bytes), since if the write fails, a ValueError will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

*Changed in version 3.6:* The number of bytes written is now returned.

**write_byte(byte)**
Write the integer byte into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.
Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the Internet.

- **email** — An email and MIME handling package
  - **email.message**: Representing an email message
  - **email.parser**: Parsing email messages
    - FeedParser API
    - Parser API
    - Additional notes
  - **email.generator**: Generating MIME documents
  - **email.policy**: Policy Objects
  - **email.errors**: Exception and Defect classes
  - **email.headerregistry**: Custom Header Objects
  - **email.contentmanager**: Managing MIME Content
    - Content Manager Instances
  - **email**: Examples
  - **email.message.Message**: Representing an email message using the compat32 API
  - **email.mime**: Creating email and MIME objects from scratch
  - **email.header**: Internationalized headers
  - **email.charset**: Representing character sets
  - **email.encoders**: Encoders
  - **email.utils**: Miscellaneous utilities
  - **email.iterators**: Iterators

- **json** — JSON encoder and decoder
  - Basic Usage
  - Encoders and Decoders
  - Exceptions
  - Standard Compliance and Interoperability
- Character Encodings
- Infinite and NaN Number Values
- Repeated Names Within an Object
- Top-level Non-Object, Non-Array Values
- Implementation Limitations
  - Command Line Interface
    - Command line options
- `mailcap` — Mailcap file handling
- `mailbox` — Manipulate mailboxes in various formats
  - Mailbox objects
    - Maildir
    - mbox
    - MH
    - Babyl
    - MMDF
  - Message objects
    - MaildirMessage
    - mboxMessage
    - MHMessage
    - BabylMessage
    - MMDFMessage
- Exceptions
- Examples
- `mimetypes` — Map filenames to MIME types
  - MimeTypes Objects
- `base64` — Base16, Base32, Base64, Base85 Data Encodings
- `binhex` — Encode and decode binhex4 files
  - Notes
- `binascii` — Convert between binary and ASCII
- `quopri` — Encode and decode MIME quoted-printable data
- `uu` — Encode and decode uuencode files
email — An email and MIME handling package

Source code: Lib/email/__init__.py

The email package is a library for managing email messages. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting RFC 5233 and RFC 6532, as well as such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, RFC 2183, and RFC 2231.

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components.

The central component of the package is an “object model” that represents email messages. An application interacts with the package primarily through the object model interface defined in the message sub-module. The application can use this API to ask questions about an existing email, to construct a new email, or to add or remove email subcomponents that themselves use the same object model interface. That is, following the nature of email messages and their MIME subcomponents, the email object model is a tree structure of objects that all provide the EmailMessage API.

The other two major components of the package are the parser and the generator. The parser takes the serialized version of an email
message (a stream of bytes) and converts it into a tree of EmailMessage objects. The generator takes an EmailMessage and turns it back into a serialized byte stream. (The parser and generator also handle streams of text characters, but this usage is discouraged as it is too easy to end up with messages that are not valid in one way or another.)

The control component is the policy module. Every EmailMessage, every generator, and every parser has an associated policy object that controls its behavior. Usually an application only needs to specify the policy when an EmailMessage is created, either by directly instantiating an EmailMessage to create a new email, or by parsing an input stream using a parser. But the policy can be changed when the message is serialized using a generator. This allows, for example, a generic email message to be parsed from disk, but to serialize it using standard SMTP settings when sending it to an email server.

The email package does its best to hide the details of the various governing RFCs from the application. Conceptually the application should be able to treat the email message as a structured tree of unicode text and binary attachments, without having to worry about how these are represented when serialized. In practice, however, it is often necessary to be aware of at least some of the rules governing MIME messages and their structure, specifically the names and nature of the MIME “content types” and how they identify multipart documents. For the most part this knowledge should only be required for more complex applications, and even then it should only be the high level structure in question, and not the details of how those structures are represented. Since MIME content types are used widely in modern internet software (not just email), this will be a familiar concept to many programmers.
The following sections describe the functionality of the `email` package. We start with the `message` object model, which is the primary interface an application will use, and follow that with the `parser` and `generator` components. Then we cover the `policy` controls, which completes the treatment of the main components of the library.

The next three sections cover the exceptions the package may raise and the defects (non-compliance with the RFCs) that the `parser` may detect. Then we cover the `headerregistry` and the `contentmanager` sub-components, which provide tools for doing more detailed manipulation of headers and payloads, respectively. Both of these components contain features relevant to consuming and producing non-trivial messages, but also document their extensibility APIs, which will be of interest to advanced applications.

Following those is a set of examples of using the fundamental parts of the APIs covered in the preceding sections.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the `Message` class, cover the legacy `compat32` API that deals much more directly with the details of how email messages are represented. The `compat32` API does not hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the `compat32` API for backward compatibility reasons.

*Changed in version 3.6:* Docs reorganized and rewritten to promote the new `EmailMessage/EmailPolicy` API.

Contents of the `email` package documentation:
- `email.message`: Representing an email message
- `email.parser`: Parsing email messages
  - FeedParser API
  - Parser API
  - Additional notes
- `email.generator`: Generating MIME documents
- `email.policy`: Policy Objects
- `email.errors`: Exception and Defect classes
- `email.headerregistry`: Custom Header Objects
- `email.contentmanager`: Managing MIME Content
  - Content Manager Instances
- `email`: Examples

Legacy API:

- `email.message.Message`: Representing an email message using the compat32 API
- `email.mime`: Creating email and MIME objects from scratch
- `email.header`: Internationalized headers
- `email.charset`: Representing character sets
- `email.encoders`: Encoders
- `email.utils`: Miscellaneous utilities
- `email.iterators`: Iterators

See also:

**Module smtplib**

SMTP (Simple Mail Transport Protocol) client

**Module poplib**

POP (Post Office Protocol) client

**Module imaplib**

IMAP (Internet Message Access Protocol) client
Module nntplib
  NNTP (Net News Transport Protocol) client

Module mailbox
  Tools for creating, reading, and managing collections of messages on disk using a variety standard formats.

Module smtplib
  SMTP server framework (primarily useful for testing)
email.message: Representing an email message

Source code: Lib/email/message.py

New in version 3.6: [1]

The central class in the `email` package is the `EmailMessage` class, imported from the `email.message` module. It is the base class for the `email` object model. `EmailMessage` provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of `headers` and a `payload` (which is also referred to as the `content`). Headers are RFC 5322 or RFC 6532 style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as `multipart/*` or `message/rfc822`.

The conceptual model provided by an `EmailMessage` object is that of an ordered dictionary of headers coupled with a `payload` that represents the RFC 5322 body of the message, which might be a list of sub-`EmailMessage` objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for
generating a serialized version of the message, and for recursively walking over the object tree.

The `EmailMessage` dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

The `payload` is either a string or bytes object, in the case of simple message objects, or a list of `EmailMessage` objects, for MIME container documents such as `multipart/*` and `message/rfc822` message objects.

```python
class email.message.EmailMessage(policy=default)

If `policy` is specified use the rules it specifies to update and serialize the representation of the message. If `policy` is not set, use the `default` policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated \r\n, it uses the Python standard \n line endings). For more information see the `policy` documentation.

as_string(unixfrom=False, maxheaderlen=None, policy=None)
```

Return the entire message flattened as a string. When optional `unixfrom` is true, the envelope header is included in the returned string. `unixfrom` defaults to `False`. For backward compatibility with the base `Message` class `maxheaderlen` is accepted, but defaults to `None`, which means that by default the line length is controlled by the `max_line_length` of the policy. The `policy`
argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified policy will be passed to the Generator.

Flattening the message may trigger changes to the EmailMessage if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See email.generator.Generator for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as “7 bit clean” when utf8 is False, which is the default.

Changed in version 3.6: the default behavior when maxheaderlen is not specified was changed from defaulting to 0 to defaulting to the value of max_line_length from the policy.

__str__()
Equivalent to
as_string(policy=self.policy.clone(utf8=True)). Allows str(msg) to produce a string containing the serialized message in a readable format.

Changed in version 3.4: the method was changed to use utf8=True, thus producing an RFC 6531-like message representation, instead of being a direct alias for as_string().
as_bytes\( (unixfrom=False, policy=None) \)
Return the entire message flattened as a bytes object. When optional \textit{unixfrom} is true, the envelope header is included in the returned string. \textit{unixfrom} defaults to \texttt{False}. The \textit{policy} argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified \textit{policy} will be passed to the \texttt{BytesGenerator}.

Flattening the message may trigger changes to the \texttt{EmailMessage} if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See \texttt{email.generator.BytesGenerator} for a more flexible API for serializing messages.

\texttt{__bytes__()}  
Equivalent to \texttt{as_bytes()}. Allows \texttt{byte(msg)} to produce a bytes object containing the serialized message.

\texttt{is_multipart()}  
Return \texttt{True} if the message’s payload is a list of sub-\texttt{EmailMessage} objects, otherwise return \texttt{False}. When \texttt{is_multipart()} returns \texttt{False}, the payload should be a string object (which might be a CTE encoded binary payload). Note that \texttt{is_multipart()} returning \texttt{True} does not necessarily mean that \texttt{“msg.get_content_maintype() == ‘multipart’”} will return the \texttt{True}. For example, \texttt{is_multipart}
will return True when the EmailMessage is of type message/rfc822.

**set_unixfrom**(unixfrom)
Set the message's envelope header to unixfrom, which should be a string. (See mboxMessage for a brief description of this header.)

**get_unixfrom**()
Return the message's envelope header. Defaults to None if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by keys(), but in an EmailMessage object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

**__len__**()
Return the total number of headers, including duplicates.
__contains__(name)
Return true if the message object has a field named name. Matching is done without regard to case and name does not include the trailing colon. Used for the in operator. For example:

```python
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

__getitem__(name)
Return the value of the named header field. name does not include the colon field separator. If the header is missing, None is returned; a KeyError is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the get_all() method to get the values of all the extant headers named name.

Using the standard (non-compat32) policies, the returned value is an instance of a subclass of email.headerregistry.BaseHeader.

__setitem__(name, val)
Add a header to the message with field name name and value val. The field is appended to the end of the message's existing headers.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name name, delete the field first, e.g.:
If the policy defines certain headers to be unique (as the standard policies do), this method may raise a `ValueError` when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency’s sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

```python
__delitem__(name)
```
Delete all occurrences of the field with name `name` from the message’s headers. No exception is raised if the named field isn’t present in the headers.

```python
keys()
```
Return a list of all the message’s header field names.

```python
values()
```
Return a list of all the message’s field values.

```python
items()
```
Return a list of 2-tuples containing all the message’s field headers and values.

```python
get(name, failobj=None)
```
Return the value of the named header field. This is identical to `__getitem__()` except that optional `failobj` is returned if the named header is missing (`failobj` defaults to `None`).

Here are some additional useful header related methods:
get_all(name, failobj=None)

Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).

add_header(_name, _value, **_params)

Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as key="value" unless the value is None, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see RFC 2231 for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in RFC 2231 format using a CHARSET of utf-8 and a LANGUAGE of None.

Here is an example:

```python
msg.add_header('Content-Disposition', 'attachment')
```
This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.
```

An example of the extended interface with non-ASCII characters:

```python
msg.add_header('Content-Disposition', 'attachment; filename=('iso-8859-1', '', 'Fußballer.ppt'
```

**replace_header(name, value)**

Replace a header. Replace the first header found in the message that matches `name`, retaining header order and field name case of the original header. If no matching header is found, raise a `KeyError`.

**get_content_type()**

Return the message's content type, coerced to lower case of the form `maintype/subtype`. If there is no `Content-Type` header in the message return the value returned by `get_default_type()`. If the `Content-Type` header is invalid, return `text/plain`.

(According to [RFC 2045](https://tools.ietf.org/html/rfc2045), messages always have a default type, `get_content_type()` will always return a value. [RFC 2045](https://tools.ietf.org/html/rfc2045) defines a message's default type to be `text/plain` unless it appears inside a `multipart/digest` container, in which case it would be `message/rfc822`. If the `Content-Type` header has an invalid type specification, [RFC 2045](https://tools.ietf.org/html/rfc2045) mandates that the default type be `text/plain`.)
get_content_maintype()
Return the message's main content type. This is the maintype part of the string returned by get_content_type().

get_content_subtype()
Return the message’s sub-content type. This is the subtype part of the string returned by get_content_type().

get_default_type()
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

set_default_type(ctype)
Set the default content type. ctype should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type header, so it only affects the return value of the get_content_type methods when no Content-Type header is present in the message.

set_param(param, value, header='Content-Type', requote=True, charset=None, language='', replace=False)
Set a parameter in the Content-Type header. If the parameter already exists in the header, replace its value with value. When header is Content-Type (the default) and the header does not yet exist in the message, add it, set its value to text/plain, and append the new parameter value. Optional header specifies an alternative header to Content-Type.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional charset
and *language* parameters. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the *utf8 charset* and None for the *language*.

If *replace* is False (the default) the header is moved to the end of the list of headers. If *replace* is True, the header will be updated in place.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the *params* attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

*Changed in version 3.4:* *replace* keyword was added.

```python
del_param(param, header='content-type', requote=True)
```
Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

```python
get_filename(failobj=None)
```
Return the value of the *filename* parameter of the *Content-Disposition* header of the message. If the header does not have a *filename* parameter, this method falls back to looking for the *name* parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The
get_boundary(failobj=None)
Return the value of the boundary parameter of the Content-Type header of the message, or failobj if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per email.utils.unquote().

set_boundary(boundary)
Set the boundary parameter of the Content-Type header to boundary. set_boundary() will always quote boundary if necessary. A HeaderParseError is raised if the message object has no Content-Type header.

Note that using this method is subtly different from deleting the old Content-Type header and adding a new one with the new boundary via add_header(), because set_boundary() preserves the order of the Content-Type header in the list of headers.

get_content_charset(failobj=None)
Return the charset parameter of the Content-Type header, coerced to lower case. If there is no Content-Type header, or if that header has no charset parameter, failobj is returned.

get_charsets(failobj=None)
Return a list containing the character set names in the message. If the message is a multipart, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.
Each item in the list will be a string which is the value of the charset parameter in the Content-Type header for the represented subpart. If the subpart has no Content-Type header, no charset parameter, or is not of the text main MIME type, then that item in the returned list will be failobj.

is_attachment()
Return True if there is a Content-Disposition header and its (case insensitive) value is attachment, False otherwise.

Changed in version 3.4.2: is_attachment is now a method instead of a property, for consistency with is_multipart().

get_content_disposition()
Return the lowercased value (without parameters) of the message's Content-Disposition header if it has one, or None. The possible values for this method are inline, attachment or None if the message follows RFC 2183.

New in version 3.5.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()
The walk() method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use walk() as the iterator in a for loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:
for part in msg.walk():
...   print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain

walk iterates over the subparts of any part where is_multipart() returns True, even though msg.get_content_maintype() == 'multipart' may return False. We can see this in our example by making use of the _structure debug helper function:

>>> for part in msg.walk():
...   print(part.get_content_maintype() == 'multipart', part.is_multipart())
True True
False False
False True
False False
False False
False True
False True
False False
>>> _structure(msg)
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
Here the message parts are not multipart, but they do contain subparts. is_multipart() returns True and walk descends into the subparts.

**get_body**(*preferencelist=('related', 'html', 'plain'))

Return the MIME part that is the best candidate to be the “body” of the message.

*preferencelist* must be a sequence of strings from the set related, html, and plain, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the get_body method is called.

If related is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a multipart/related, check the start parameter and if a part with a matching Content-ID is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the multipart/related.

If a part has a Content-Disposition header, only consider the part a candidate match if the value of the header is inline.

If none of the candidates matches any of the preferences in *preferencelist*, return None.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are ('plain',),
('html', 'plain'), and the default ('related', 'html', 'plain').

(2) Because matching starts with the object on which get_body is called, calling get_body on a multipart/related will return the object itself unless preference list has a non-default value. (3) Messages (or message parts) that do not specify a Content-Type or whose Content-Type header is invalid will be treated as if they are of type text/plain, which may occasionally cause get_body to return unexpected results.

iter_attachments()
Return an iterator over all of the immediate sub-parts of the message that are not candidate “body” parts. That is, skip the first occurrence of each of text/plain, text/html, multipart/related, or multipart/alternative (unless they are explicitly marked as attachments via Content-Disposition: attachment), and return all remaining parts. When applied directly to a multipart/related, return an iterator over the all the related parts except the root part (ie: the part pointed to by the start parameter, or the first part if there is no start parameter or the start parameter doesn’t match the Content-ID of any of the parts). When applied directly to a multipart/alternative or a non-multipart, return an empty iterator.

iter_parts()
Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also walk()).

get_content(*args, content_manager=None, **kw)
Call the `get_content()` method of the `content_manager`, passing `self` as the message object, and passing along any other arguments or keywords as additional arguments. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`.

`set_content(*args, content_manager=None, **kw)`
Call the `set_content()` method of the `content_manager`, passing `self` as the message object, and passing along any other arguments or keywords as additional arguments. If `content_manager` is not specified, use the `content_manager` specified by the current `policy`.

`make_related(boundary=None)`
Convert a non-multipart message into a `multipart/related` message, moving any existing `Content`-headers and payload into a (new) first part of the `multipart`. If `boundary` is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

`make_alternative(boundary=None)`
Convert a non-multipart or a `multipart/related` into a `multipart/alternative`, moving any existing `Content`-headers and payload into a (new) first part of the `multipart`. If `boundary` is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

`make_mixed(boundary=None)`
Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing Content- headers and payload into a (new) first part of the multipart. If boundary is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

**add_related(** *args, content_manager=None, **kw**)

If the message is a multipart/related, create a new message object, pass all of the arguments to its set_content() method, and attach() it to the multipart. If the message is a non-multipart, call make_related() and then proceed as above. If the message is any other type of multipart, raise a TypeError. If content_manager is not specified, use the content_manager specified by the current policy. If the added part has no Content-Disposition header, add one with the value inline.

**add_alternative(** *args, content_manager=None, **kw**)

If the message is a multipart/alternative, create a new message object, pass all of the arguments to its set_content() method, and attach() it to the multipart. If the message is a non-multipart or multipart/related, call make_alternative() and then proceed as above. If the message is any other type of multipart, raise a TypeError. If content_manager is not specified, use the content_manager specified by the current policy.

**add_attachment(** *args, content_manager=None, **kw**)

If the message is a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing Content- headers and payload into a (new) first part of the multipart. If boundary is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).
If the message is a multipart/mixed, create a new message object, pass all of the arguments to its set_content() method, and attach() it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call make_mixed() and then proceed as above. If content_manager is not specified, use the content_manager specified by the current policy. If the added part has no Content-Disposition header, add one with the value attachment. This method can be used both for explicit attachments (Content-Disposition: attachment) and inline attachments (Content-Disposition: inline), by passing appropriate options to the content_manager.

clear()
Remove the payload and all of the headers.

clear_content()
Remove the payload and all of the Content- headers, leaving all other headers intact and in their original order.

EmailMessage objects have the following instance attributes:

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.
The *preamble* attribute contains this leading extra-armor text for MIME documents. When the **Parser** discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the **Generator** is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

**epilogue**

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be `None`.

**defects**

The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

**class email.message.MIMEPart**(policy=default)

This class represents a subpart of a MIME message. It is identical to `EmailMessage`, except that no *MIME-Version* headers are added when `set_content()` is called, since sub-parts do not need their own *MIME-Version* headers.

**Footnotes**
Originally added in 3.4 as a provisional module. Docs for legacy message class moved to `email.message.Message`: Representing an email message using the compat32 API.
email — An email and MIME handling package

Internet Data Handling

The Python Standard Library

previous | next | modules | index
email.parser: Parsing email messages

Source code: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an EmailMessage object, adding headers using the dictionary interface, and adding payload(s) using set_content() and related methods, or they can be created by parsing a serialized representation of the email message.

The email package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root EmailMessage instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return True from its is_multipart() method, and the subparts can be accessed via the payload manipulation methods, such as get_body(), iter_parts(), and walk().

There are actually two parser interfaces available for use, the Parser API and the incremental FeedParser API. The Parser API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. FeedParser is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The FeedParser can consume and parse the
message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the email package's bundled parser and the EmailMessage class is embodied in the policy class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate policy methods.
FeedParser API

The **BytesFeedParser**, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The **BytesFeedParser** can of course be used to parse an email message fully contained in a bytes-like object, string, or file, but the **BytesParser** API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The **BytesFeedParser**’s API is simple; you create an instance, feed it a bunch of bytes until there’s no more to feed it, then close the parser to retrieve the root message object. The **BytesFeedParser** is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object’s `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Here is the API for the **BytesFeedParser**:

```python
class email.parser.BytesFeedParser(_factory=None, *, policy=policy.compat32)
```

Create a **BytesFeedParser** instance. Optional `_factory` is a no-argument callable; if not specified use the `message_factory` from the `policy`. Call `_factory` whenever a new message object is needed.

If `policy` is specified use the rules it specifies to update the
representation of the message. If policy is not set, use the compat32 policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides Message as the default factory. All other policies provide EmailMessage as the default _factory. For more information on what else policy controls, see the policy documentation.

Note: The policy keyword should always be specified; The default will change to email.policy.default in a future version of Python.

New in version 3.2.

Changed in version 3.3: Added the policy keyword.

Changed in version 3.6: _factory defaults to the policy message_factory.

feed(data)
Feed the parser some more data. data should be a bytes-like object containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()
Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if feed() is called after this method has been called.

class email.parser.FeedParser(_factory=None, *, policy=policy.compat32)
Works like BytesFeedParser except that the input to the feed() method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if utf8 is True, no binary attachments.

*Changed in version 3.3:* Added the *policy* keyword.
Parser API

The **BytesParser** class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a bytes-like object or file. The `email.parser` module also provides **Parser** for parsing strings, and header-only parsers, **BytesHeaderParser** and **HeaderParser**, which can be used if you're only interested in the headers of the message. **BytesHeaderParser** and **HeaderParser** can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

```python
class email.parser.BytesParser(_class=None, *, policy=policy.compat32)
```

Create a **BytesParser** instance. The `_class` and `policy` arguments have the same meaning and semantics as the `_factory` and `policy` arguments of **BytesFeedParser**.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

*Changed in version 3.3:* Removed the `strict` argument that was deprecated in 2.4. Added the `policy` keyword.

*Changed in version 3.6:* `_class` defaults to the policy `message_factory`.

```python
parse(fp, headersonly=False)
```

Read all the data from the binary file-like object `fp`, parse the resulting bytes, and return the message object. `fp` must support
both the `readline()` and the `read()` methods.

The bytes contained in `fp` must be formatted as a block of RFC 5322 (or, if `utf8` is True, RFC 6532) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a `Content-Transfer-Encoding` of 8bit).

Optional `headersonly` is a flag specifying whether to stop parsing after reading the headers or not. The default is False, meaning it parses the entire contents of the file.

**parsebytes**(bytes, headersonly=False)

Similar to the `parse()` method, except it takes a bytes-like object instead of a file-like object. Calling this method on a bytes-like object is equivalent to wrapping `bytes` in a `BytesIO` instance first and calling `parse()`.

Optional `headersonly` is as with the `parse()` method.

*New in version 3.2.*

```python
class email.parser.BytesHeaderParser(_class=None,*,
policy=policy.compat32)
```

Exactly like `BytesParser`, except that `headersonly` defaults to True.

*New in version 3.3.*

```python
class email.parser.Parser(_class=None,*,
policy=policy.compat32)
```
This class is parallel to BytesParser, but handles string input.

*Changed in version 3.3:* Removed the `strict` argument. Added the `policy` keyword.

*Changed in version 3.6:* `_class` defaults to the policy `message_factory`.

**parse**(*fp, headersonly=False*)

Read all the data from the text-mode file-like object `fp`, parse the resulting text, and return the root message object. `fp` must support both the `readline()` and the `read()` methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

**parsestr**(*text, headersonly=False*)

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping `text` in a `StringIO` instance first and calling `parse()`.

Optional `headersonly` is as with the `parse()` method.

```python
class email.parser/HeaderParser(_class=None, *, policy=policy.compat32)
    Exactly like Parser, except that `headersonly` defaults to `True`.
```

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.
email.message_from_bytes(s, _class=None, *,
policy=policy.compat32)
    Return a message object structure from a bytes-like object. This is equivalent to BytesParser().parsebytes(s). Optional _class and policy are interpreted as with the BytesParser class constructor.

    New in version 3.2.

    Changed in version 3.3: Removed the strict argument. Added the policy keyword.

email.message_from_binary_file(fp, _class=None, *,
policy=policy.compat32)
    Return a message object structure tree from an open binary file object. This is equivalent to BytesParser().parse(fp). _class and policy are interpreted as with the BytesParser class constructor.

    New in version 3.2.

    Changed in version 3.3: Removed the strict argument. Added the policy keyword.

email.message_from_string(s, _class=None, *,
policy=policy.compat32)
    Return a message object structure from a string. This is equivalent to Parser().parsestr(s). _class and policy are interpreted as with the Parser class constructor.

    Changed in version 3.3: Removed the strict argument. Added the policy keyword.
email.message_from_file(fp, _class=None, *, policy=policy.compat32)

Return a message object structure tree from an open file object. This is equivalent to Parser().parse(fp). _class and policy are interpreted as with the Parser class constructor.

Changed in version 3.3: Removed the strict argument. Added the policy keyword.

Changed in version 3.6: _class defaults to the policy message_factory.

Here's an example of how you might use message_from_bytes() at an interactive Python prompt:

```python
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```
Additional notes

Here are some notes on the parsing semantics:

- Most non-\texttt{multipart} type messages are parsed as a single message object with a string payload. These objects will return \texttt{False} for \texttt{is\_multipart()}, and \texttt{iter\_parts()} will yield an empty list.
- All \texttt{multipart} type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return \texttt{True} for \texttt{is\_multipart()}, and \texttt{iter\_parts()} will yield a list of subparts.
- Most messages with a content type of \texttt{message/*} (such as \texttt{message/delivery-status} and \texttt{message/rfc822}) will also be parsed as container object containing a list payload of length 1. Their \texttt{is\_multipart()} method will return \texttt{True}. The single element yielded by \texttt{iter\_parts()} will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their \texttt{multipart}-edness. Such messages may have a \texttt{Content-Type} header of type \texttt{multipart}, but their \texttt{is\_multipart()} method may return \texttt{False}. If such messages were parsed with the \texttt{FeedParser}, they will have an instance of the \texttt{MultipartInvariantViolationDefect} class in their \texttt{defects} attribute list. See \texttt{email.errors} for details.
email.generator: Generating MIME documents

Source code: Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via smtplib.SMTP.sendmail() or the nntplib module, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the email.parser module, you aren’t limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming policy is used for both. That is, parsing the serialized byte stream via the BytesParser class and then regenerating the serialized byte stream using BytesGenerator should produce output identical to the input [1]. (On the other hand, using the generator on an EmailMessage constructed by program may result in changes to the EmailMessage object as defaults are filled in.)

The Generator class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode
cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not “8 bit clean”.

```python
class email.generator.BytesGenerator(outfp, mangle_from_=None, maxheaderlen=None, *, policy=None)
    Return a BytesGenerator object that will write any message provided to the flatten() method, or any surrogateescape encoded text provided to the write() method, to the file-like object outfp. outfp must support a write method that accepts binary data.

    If optional mangle_from_ is True, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. mangle_from_ defaults to the value of the mangle_from_ setting of the policy (which is True for the compat32 policy and False for all others). mangle_from_ is intended for use when messages are stored in unix mbox format (see mailbox and WHY THE CONTENT-LENGTH FORMAT IS BAD).

    If maxheaderlen is not None, refold any header lines that are longer than maxheaderlen, or if 0, do not rewrap any headers. If maxheaderlen is None (the default), wrap headers and other message lines according to the policy settings.

    If policy is specified, use that policy to control message generation. If policy is None (the default), use the policy associated with the Message or EmailMessage object passed to flatten to control the message generation. See email.policy for details on what
```
policy controls.

New in version 3.2.

Changed in version 3.3: Added the policy keyword.

Changed in version 3.6: The default behavior of the mangle_from_ and maxheaderlen parameters is to follow the policy.

flatten(msg, unixfrom=False, linesep=None)

Print the textual representation of the message object structure rooted at msg to the output file specified when the BytesGenerator instance was created.

If the policy option cte_type is 8bit (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII Content-Transfer-Encoding of any body parts that have them. If cte_type is 7bit, convert the bytes with the high bit set as needed using an ASCII-compatible Content-Transfer-Encoding. That is, transform parts with non-ASCII Content-Transfer-Encoding (Content-Transfer-Encoding: 8bit) to an ASCII compatible Content-Transfer-Encoding, and encode RFC-invalid non-ASCII bytes in headers using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

If unixfrom is True, print the envelope header delimiter used by the Unix mailbox format (see mailbox) before the first of the RFC 5322 headers of the root message object. If the root object has no envelope header, craft a standard one. The default is False. Note that for subparts, no envelope header is ever printed.
If `linesep` is not `None`, use it as the separator character between all the lines of the flattened message. If `linesep` is `None` (the default), use the value specified in the `policy`.

**clone**(fp)

Return an independent clone of this `BytesGenerator` instance with the exact same option settings, and `fp` as the new `outfp`.

**write**(s)

Encode `s` using the ASCII codec and the `surrogateescape` error handler, and pass it to the `write` method of the `outfp` passed to the `BytesGenerator`'s constructor.

As a convenience, `EmailMessage` provides the methods `as_bytes()` and `bytes(aMessage)` (a.k.a. `__bytes__()`), which simplify the generation of a serialized binary representation of a message object. For more detail, see `email.message`.

Because strings cannot represent binary data, the `Generator` class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible `Content-Transfer_Encoding`. Using the terminology of the email RFCs, you can think of this as `Generator` serializing to an I/O stream that is not “8 bit clean”. In other words, most applications will want to be using `BytesGenerator`, and not `Generator`.

```python
class email.generator.**Generator**(outfp,
mangle_from_=None, maxheaderlen=None, *, policy=None)
```

Return a `Generator` object that will write any message provided to the `flatten()` method, or any text provided to the `write()` method, to the file-like object `outfp`. `outfp` must support a `write`
method that accepts string data.

If optional `mangle_from_` is True, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. `mangle_from_` defaults to the value of the `mangle_from_` setting of the `policy` (which is True for the `compat32` policy and False for all others). `mangle_from_` is intended for use when messages are stored in unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If `maxheaderlen` is not None, refold any header lines that are longer than `maxheaderlen`, or if 0, do not rewrap any headers. If `maxheaderlen` is None (the default), wrap headers and other message lines according to the `policy` settings.

If `policy` is specified, use that policy to control message generation. If `policy` is None (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what `policy` controls.

*Changed in version 3.3:* Added the `policy` keyword.

*Changed in version 3.6:* The default behavior of the `mangle_from_` and `maxheaderlen` parameters is to follow the policy.

```python
flatten(msg, unixfrom=False, linesep=None)
```

Print the textual representation of the message object structure rooted at `msg` to the output file specified when the `Generator` instance was created.

If the `policy` option `cte_type` is 8bit, generate the
message as if the option were set to 7bit. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible Content-Transfer-Encoding. That is, transform parts with non-ASCII Content-Transfer-Encoding (Content-Transfer-Encoding: 8bit) to an ASCII compatible Content-Transfer-Encoding, and encode RFC-invalid non-ASCII bytes in headers using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

If unixfrom is True, print the envelope header delimiter used by the Unix mailbox format (see mailbox) before the first of the RFC 5322 headers of the root message object. If the root object has no envelope header, craft a standard one. The default is False. Note that for subparts, no envelope header is ever printed.

If linesep is not None, use it as the separator character between all the lines of the flattened message. If linesep is None (the default), use the value specified in the policy.

*Changed in version 3.2:* Added support for re-encoding 8bit message bodies, and the linesep argument.

**clone**(fp)

Return an independent clone of this Generator instance with the exact same options, and fp as the new outfp.

**write**(s)

Write s to the write method of the outfp passed to the Generator's constructor. This provides just enough file-like API for Generator instances to be used in the print()
function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-text parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

```python
class email.generator. DecodedGenerator(outfp, mangle_from_=None, maxheaderlen=None, fmt=None, *, policy=None)
```

Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type `text`, print the decoded payload of the subpart, and if the main type is not `text`, instead of printing it fill in the string `fmt` using information from the part and print the resulting filled-in string.

To fill in `fmt`, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` – Full MIME type of the non-text part
- `maintype` – Main MIME type of the non-text part
- `subtype` – Sub-MIME type of the non-text part
- `filename` – Filename of the non-text part
- `description` – Description associated with the non-text part
- `encoding` – Content transfer encoding of the non-text part

If `fmt` is `None`, use the following default `fmt`:
“[Non-text (%(type)s) part of message omitted, filename % (filename)s]”

Optional _mangle_from_ and maxheaderlen are as with the Generator base class.

Footnotes

This statement assumes that you use the appropriate setting for unixfrom, and that there are no policy settings calling for automatic adjustments (for example, refold_source must be none, which is not the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

[1]
The `email` package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A `Policy` object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. `Policy` instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the `parser` classes and the related convenience functions, and for the `Message` class, this is the `Compat32` policy, via its
corresponding pre-defined instance compat32. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the policy keyword to EmailMessage is the EmailPolicy policy, via its pre-defined instance default.

When a Message or EmailMessage object is created, it acquires a policy. If the message is created by a parser, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a generator, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the policy keyword for the email.parser classes and the parser convenience functions will be changing in a future version of Python. Therefore you should always specify explicitly which policy you want to use when calling any of the classes and functions described in the parser module.

The first part of this documentation covers the features of Policy, an abstract base class that defines the features that are common to all policy objects, including compat32. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes EmailPolicy and Compat32, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

Policy instances are immutable, but they can be cloned, accepting
the same keyword arguments as the class constructor and returning a new `Policy` instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```python
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy)
>>> p = Popen(['sendmail', msg['To'].addresses[0]],
...             stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone)
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling `BytesGenerator` to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a `policy` keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the `as_bytes()` method of the `msg` object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```python
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone))
```
Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```python
>>> compat_SMTP = policy.compat32.clone(linesep='
')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```python
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length  # 80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length  # 100
```

class email.policy.Policy(**kw)

This is the abstract base class for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the clone() method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the
constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

**max_line_length**

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per RFC 5322. A value of 0 or None indicates that no line wrapping should be done at all.

**linesep**

The string to be used to terminate lines in serialized output. The default is \n because that’s the internal end-of-line discipline used by Python, though \r\n is required by the RFCs.

**cte_type**

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

<table>
<thead>
<tr>
<th>cte_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7bit</td>
<td>all data must be “7 bit clean” (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.</td>
</tr>
<tr>
<td>8bit</td>
<td>data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see fold_binary() and utf8 below for exceptions), but body parts may use the 8bit CTE.</td>
</tr>
</tbody>
</table>

A cte_type value of 8bit only works with BytesGenerator, not Generator, because strings cannot contain binary data. If a Generator is operating under a policy
that specifies `cte_type=8bit`, it will act as if `cte_type` is 7bit.

**raise_on_defect**
If `True`, any defects encountered will be raised as errors. If `False` (the default), defects will be passed to the `register_defect()` method.

**mangle_from_**
If `True`, lines starting with "From " in the body are escaped by putting a `>` in front of them. This parameter is used when the message is being serialized by a generator. Default: `False`.

*New in version 3.5:* The `mangle_from_` parameter.

**message_factory**
A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case `Message` is used.

*New in version 3.6.*

The following `Policy` method is intended to be called by code using the email library to create policy instances with custom settings:

**clone(**kw**)
Return a new `Policy` instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining `Policy` methods are called by the email package code, and are not intended to be called by an application using the
email package. A custom policy must implement all of these methods.

**handle_defect**(obj, defect)
Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of Defect.

The default implementation checks the `raise_on_defect` flag. If it is True, *defect* is raised as an exception. If it is False (the default), *obj* and *defect* are passed to `register_defect()`.

**register_defect**(obj, defect)
Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of Defect.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls `handle_defect`, *obj* will normally have a `defects` attribute that has an append method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

**header_max_count**(name)
Return the maximum allowed number of headers named *name*.

Called when a header is added to an `EmailMessage` or `Message` object. If the returned value is not 0 or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a `ValueError` is raised.
Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

**header_source_parse** *(sourcelines)*

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, `name` should be the case preserved name (all characters up to the `:` separator), while `value` should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

`sourcelines` may contain surrogateescaped binary data.

There is no default implementation

**header_store_parse** *(name, value)*

The email package calls this method with the name and value provided by the application program when the application
program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the `name` and `value` should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

**header_fetch_parse**(name, value)

The email package calls this method with the `name` and `value` currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

`value` may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

**fold**(name, value)

The email package calls this method with the `name` and `value` currently stored in the `Message` for a given header. The method should return a string that represents that header “folded” correctly (according to the policy settings) by
composing the name with the value and inserting linesep characters at the appropriate places. See RFC 5322 for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(name, value)
The same as fold(), except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.EmailPolicy(**kw)
This concrete Policy provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) RFC 5322, RFC 2047, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are str subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement RFC 2047 and RFC 5322.

The default value for the message_factory attribute is EmailMessage.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

New in version 3.6: [1]

utf8
If False, follow RFC 5322, supporting non-ASCII characters in headers by encoding them as “encoded words”. If True, follow RFC 6532 and use utf-8 encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the SMTPUTF8 extension (RFC 6531).

refold_source

If the value for a header in the Message object originated from a parser (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

<table>
<thead>
<tr>
<th>None</th>
<th>All source values use original folding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>Source values that have any line that is longer than max_line_length will be refolded</td>
</tr>
<tr>
<td>All</td>
<td>All values are refolded.</td>
</tr>
</tbody>
</table>

The default is Long.

header_factory

A callable that takes two arguments, name and value, where name is a header field name and value is an unfolded header field value, and returns a string subclass that represents that header. A default header_factory (see headerregistry) is provided that supports custom parsing for the various address and date RFC 5322 header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager

An object with at least two methods: get_content and
set_content. When the `get_content()` or `set_content()` method of an `EmailMessage` object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to `raw_data_manager`.

*New in version 3.4.*

The class provides the following concrete implementations of the abstract methods of `Policy`:

**header_max_count**(name)

Returns the value of the `max_count` attribute of the specialized class used to represent the header with the given name.

**header_source_parse**(sourcelines)

The name is parsed as everything up to the `:` and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

**header_store_parse**(name, value)

The name is returned unchanged. If the input value has a `name` attribute and it matches `name` ignoring case, the value is returned unchanged. Otherwise the `name` and `value` are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

**header_fetch_parse**(name, value)
If the value has a `name` attribute, it is returned to unmodified. Otherwise the `name`, and the `value` with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

**fold**(name, value)

Header folding is controlled by the `refold_source` policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the `name` and the `value` with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using `splitlines()`. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

**fold_binary**(name, value)

The same as `fold()` if `cte_type` is 7bit, except that the returned value is bytes.

If `cte_type` is 8bit, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no
way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of `EmailPolicy` provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the HTTP instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

**email.policy.default**

An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

**email.policy.SMTP**

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

**email.policy.SMTPUTF8**

The same as `SMTP` except that `utf8` is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the `smtplib.SMTP.send_message()` method handles this automatically).

**email.policy.HTTP**

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

**email.policy.strict**

Convenience instance. The same as `default` except that
raise_on_defect is set to True. This allows any policy to be made strict by writing:

```python
somepolicy + policy.strict
```

With all of these EmailPolicies, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a `Message` results in that header being parsed and a header object created.
- Fetching a header value from a `Message` results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the `EmailMessage` is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in `headerregistry`.

```python
class email.policyCompat32(**kw)
```

This concrete `Policy` is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The `policy` module also defines an instance of this class, `compat32`, that is used as the default policy. Thus the default behavior of the
email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the Policy default:

mangle_from_

The default is True.

The class provides the following concrete implementations of the abstract methods of Policy:

header_source_parse(sourcelines)
The name is parsed as everything up to the ‘:’ and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(name, value)
The name and value are returned unmodified.

header_fetch_parse(name, value)
If the value contains binary data, it is converted into a Header object using the unknown-8bit charset. Otherwise it is returned unmodified.

fold(name, value)
Headers are folded using the Header folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the max_line_length. Non-ASCII binary data are CTE encoded using the unknown-8bit charset.

fold_binary(name, value)
Headers are folded using the `Header` folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

`email.policy.compat32`

An instance of `Compat32`, providing backward compatibility with the behavior of the email package in Python 3.2.

**Footnotes**

[1] Originally added in 3.3 as a `provisional feature`.
email.errors: Exception and Defect classes

Source code: Lib/email/errors.py

The following exception classes are defined in the email.errors module:

*exception* email.errors.MessageError

This is the base class for all exceptions that the email package can raise. It is derived from the standard Exception class and defines no additional methods.

*exception* email.errors.MessageParseError

This is the base class for exceptions raised by the Parser class. It is derived from MessageError. This class is also used internally by the parser used by headerregistry.

*exception* email.errors.HeaderParseError

Raised under some error conditions when parsing the RFC 5322 headers of a message, this class is derived from MessageParseError. The set_boundary() method will raise this error if the content type is unknown when the method is called. Header may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).
exception email.errors.BoundaryError

Deprecated and no longer used.

exception email.errors.MultipartConversionError

Raised when a payload is added to a Message object using add_payload(), but the payload is already a scalar and the message’s Content-Type main type is not either multipart or missing. MultipartConversionError multiply inherits from MessageError and the built-in TypeError.

Since Message.add_payload() is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the attach() method is called on an instance of a class derived from MIMENonMultipart (e.g. MIMEImage).

Here is the list of the defects that the FeedParser can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a multipart/alternative had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from email.errors.MessageDefect.

- NoBoundaryInMultipartDefect – A message claimed to be a multipart, but had no boundary parameter.

- StartBoundaryNotFoundDefect – The start boundary claimed in the Content-Type header was never found.

- CloseBoundaryNotFoundDefect – A start boundary was found, but no corresponding close boundary was ever found.
New in version 3.3.

- **FirstHeaderLineIsContinuationDefect** – The message had a continuation line as its first header line.

- **MisplacedEnvelopeHeaderDefect** - A “Unix From” header was found in the middle of a header block.

- **MissingHeaderBodySeparatorDefect** - A line was found while parsing headers that had no leading white space but contained no ‘:’. Parsing continues assuming that the line represents the first line of the body.

New in version 3.3.

- **MalformedHeaderDefect** – A header was found that was missing a colon, or was otherwise malformed.

  *Deprecated since version 3.3:* This defect has not been used for several Python versions.

- **MultipartInvariantViolationDefect** – A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its `is_multipart()` method may return false even though its content type claims to be *multipart*.

- **InvalidBase64PaddingDefect** – When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.

- **InvalidBase64CharactersDefect** – When decoding a block of base64 encoded bytes, characters outside the base64 alphabet
were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.

- **InvalidBase64LengthDefect** – When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.
email — An email and MIME handling package
email.headerregistry: Custom Header Objects

Source code: Lib/email/headerregistry.py

New in version 3.6: [1]

Headers are represented by customized subclasses of str. The particular class used to represent a given header is determined by the header_factor of the policy in effect when the headers are created. This section documents the particular header_factor implemented by the email package for handling RFC 5322 compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from EmailPolicy, all headers are produced by HeaderRegistry and have BaseHeader as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class UnstructuredHeader as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the HeaderRegistry. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of


**HeaderRegistry**, and finally the support classes used to represent the data parsed from structured headers.

```python
class email.headerregistry.BaseHeader(name, value)
    name and value are passed to BaseHeader from the header_factory call. The string value of any header object is the value fully decoded to unicode.
```

This base class defines the following read-only properties:

**name**

The name of the header (the portion of the field before the `:`). This is exactly the value passed in the header_factory call for name; that is, case is preserved.

**defects**

A tuple of HeaderDefect instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the errors module for a discussion of the types of defects that may be reported.

**max_count**

The maximum number of headers of this type that can have the same name. A value of None means unlimited. The BaseHeader value for this attribute is None; it is expected that specialized header classes will override this value as needed.

BaseHeader also provides the following method, which is called by the email library code and should not in general be called by application programs:

```python
fold(*, policy)
```
Return a string containing \texttt{linesep} characters as required to correctly fold the header according to \textit{policy}. A \texttt{cte\_type} of 8bit will be treated as if it were 7bit, since headers may not contain arbitrary binary data. If \texttt{utf8} is \texttt{False}, non-ASCII data will be \texttt{RFC 2047} encoded.

\texttt{BaseHeader} by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, \texttt{BaseHeader} requires that the specialized class provide a \texttt{classmethod()} named \texttt{parse}. This method is called as follows:

\begin{verbatim}
parse(string, kwds)
\end{verbatim}

\texttt{kwds} is a dictionary containing one pre-initialized key, \texttt{defects}. \texttt{defects} is an empty list. The \texttt{parse} method should append any detected defects to this list. On return, the \texttt{kwds} dictionary \texttt{must} contain values for at least the keys \texttt{decoded} and \texttt{defects}. \texttt{decoded} should be the string value for the header (that is, the header value fully decoded to unicode). The \texttt{parse} method should assume that \texttt{string} may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

\texttt{BaseHeader}'s \texttt{\_new\_} then creates the header instance, and calls its \texttt{init} method. The specialized class only needs to provide an \texttt{init} method if it wishes to set additional attributes beyond those provided by \texttt{BaseHeader} itself. Such an \texttt{init} method should look like this:

\begin{verbatim}
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
\end{verbatim}
That is, anything extra that the specialized class puts in to the 
kwds dictionary should be removed and handled, and the 
remaining contents of kw (and args) passed to the BaseHeader 
init method.

```
super().init(*args, **kw)
```

```python
class email.headerregistry.UnstructuredHeader

An “unstructured” header is the default type of header in RFC 5322. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the Subject header.

In RFC 5322, an unstructured header is a run of arbitrary text in the ASCII character set. RFC 2047, however, has an RFC 5322 compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a value containing encoded words is passed to the constructor, the UnstructuredHeader parser converts such encoded words into unicode, following the RFC 2047 rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

```python
class email.headerregistry.DateHeader

RFC 5322 specifies a very specific format for dates within email headers. The DateHeader parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found “in the wild”.
```
This header type provides the following additional attributes:

**datetime**

If the header value can be recognized as a valid date of one form or another, this attribute will contain a `datetime` instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then `datetime` will be a naive `datetime`. If a specific timezone offset is found (including `+0000`), then `datetime` will contain an aware `datetime` that uses `datetime.timezone` to record the timezone offset.

The decoded value of the header is determined by formatting the `datetime` according to the [RFC 5322](https://tools.ietf.org/html/rfc5322) rules; that is, it is set to:

```python
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, `value` may be `datetime` instance. This means, for example, that the following code is valid and does what one would expect:

```python
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the `localtime()` function from the `utils` module:

```python
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date
using the current timezone offset.

class email.headerregistry.AddressHeader
Address headers are one of the most complex structured header types. The AddressHeader class provides a generic interface to any address header.

This header type provides the following additional attributes:

**groups**
A tuple of Group objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address Groups whose display_name is None.

**addresses**
A tuple of Address objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is “flattened” into a one-dimensional list).

The decoded value of the header will have all encoded words decoded to unicode. idna encoded domain names are also decoded to unicode. The decoded value is set by joining the str value of the elements of the groups attribute with ',', '.

A list of Address and Group objects in any combination may be used to set the value of an address header. Group objects whose display_name is None will be interpreted as single addresses, which allows an address list to be copied with groups intact by
using the list obtained from the \texttt{groups} attribute of the source header.

\texttt{class email.headerregistry. SingleAddressHeader}

A subclass of \texttt{AddressHeader} that adds one additional attribute:

\textbf{address}

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default \texttt{policy}), accessing this attribute will result in a \texttt{ValueError}.

Many of the above classes also have a \texttt{Unique} variant (for example, \texttt{UniqueUnstructuredHeader}). The only difference is that in the \texttt{Unique} variant, \texttt{max_count} is set to 1.

\texttt{class email.headerregistry. MIMEVersionHeader}

There is really only one valid value for the \texttt{MIME-Version} header, and that is 1.0. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per \texttt{RFC 2045}, then the header object will have non-\texttt{None} values for the following attributes:

\textbf{version}

The version number as a string, with any whitespace and/or comments removed.

\textbf{major}

The major version number as an integer

\textbf{minor}

The minor version number as an integer
class email.headerregistry.ParameterizedMIMEHeader

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

**params**

A dictionary mapping parameter names to parameter values.

class email.headerregistry.ContentTypeHeader

A ParameterizedMIMEHeader class that handles the Content-Type header.

**content_type**

The content type string, in the form maintype/subtype.

**maintype**

**subtype**

class email.headerregistry.ContentDispositionHeader

A ParameterizedMIMEHeader class that handles the Content-Disposition header.

**content-disposition**

inline and attachment are the only valid values in common use.

class email.headerregistry.ContentTransferEncoding

Handles the Content-Transfer-Encoding header.
Valid values are 7bit, 8bit, base64, and quoted-printable. See RFC 2045 for more information.

class
e-mail.headerregistry.HeaderRegistry(base_class=BaseH
default_class=UnstructuredHeader, use_default_map=True)

This is the factory used by EmailPolicy by default. HeaderRegistry builds the class used to create a header instance dynamically, using base_class and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by default_class is used as the specialized class. When use_default_map is True (the default), the standard mapping of header names to classes is copied into the registry during initialization. base_class is always the last class in the generated class’s __bases__ list.

The default mappings are:

<table>
<thead>
<tr>
<th>Header</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject</td>
<td>UniqueUnstructuredHeader</td>
</tr>
<tr>
<td>date</td>
<td>UniqueDateHeader</td>
</tr>
<tr>
<td>resent-date</td>
<td>DateHeader</td>
</tr>
<tr>
<td>orig-date</td>
<td>UniqueDateHeader</td>
</tr>
<tr>
<td>sender</td>
<td>UniqueSingleAddressHeader</td>
</tr>
<tr>
<td>resent-sender</td>
<td>SingleAddressHeader</td>
</tr>
<tr>
<td>to</td>
<td>UniqueAddressHeader</td>
</tr>
<tr>
<td>resent-to</td>
<td>AddressHeader</td>
</tr>
<tr>
<td>cc</td>
<td>UniqueAddressHeader</td>
</tr>
<tr>
<td>resent-cc</td>
<td>AddressHeader</td>
</tr>
<tr>
<td>from</td>
<td>UniqueAddressHeader</td>
</tr>
</tbody>
</table>
resent-from:  AddressHeader
reply-to:    UniqueAddressHeader

HeaderRegistry has the following methods:

map_to_type(self, name, cls)
    name is the name of the header to be mapped. It will be converted to lower case in the registry. cls is the specialized class to be used, along with base_class, to create the class used to instantiate headers that match name.

__getitem__(name)
    Construct and return a class to handle creating a name header.

__call__(name, value)
    Retrieves the specialized header associated with name from the registry (using default_class if name does not appear in the registry) and composes it with base_class to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.Address(display_name='',
                  username='', domain='', addr_spec=None)
    The class used to represent an email address. The general form of an address is:

    [display_name] <username@domain>
or:

```plaintext
username@domain
```

where each part must conform to specific syntax rules spelled out in RFC 5322.

As a convenience `addr_spec` can be specified instead of `username` and `domain`, in which case `username` and `domain` will be parsed from the `addr_spec`. An `addr_spec` must be a properly RFC quoted string; if it is not Address will raise an error. Unicode characters are allowed and will be properly encoded when serialized. However, per the RFCs, unicode is not allowed in the username portion of the address.

**display_name**

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

**username**

The username portion of the address, with all quoting removed.

**domain**

The domain portion of the address.

**addr_spec**

The username@domain portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

```python
__str__()
```
The str value of the object is the address quoted according to RFC 5322 rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (RFC 5321), Address handles one special case: if username and domain are both the empty string (or None), then the string value of the Address is <>.

class email.headerregistry.Group(display_name=None, addresses=None)
The class used to represent an address group. The general form of an address group is:

display_name: [address-list];

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a Group may also be used to represent single addresses that are not part of a group by setting display_name to None and providing a list of the single address as addresses.

display_name
The display_name of the group. If it is None and there is exactly one Address in addresses, then the Group represents a single address that is not in a group.

addresses
A possibly empty tuple of Address objects representing the addresses in the group.

__str__()
The str value of a Group is formatted according to RFC 5322,
but with no Content Transfer Encoding of any non-ASCII characters. If display_name is none and there is a single Address in the addresses list, the str value will be the same as the str of that single Address.

Footnotes

[1] Originally added in 3.3 as a provisional module
email — An email and MIME handling package
email.contentmanager: Managing MIME Content

Source code: Lib/email/contentmanager.py

New in version 3.6: [1]

class email.contentmanager.ContentManager
    Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the get_content and set_content dispatch methods.

get_content(msg, *args, **kw)
    Look up a handler function based on the mimetype of msg (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from msg and return an object that encodes information about the extracted data.

    To find the handler, look for the following keys in the registry, stopping with the first one found:

        • the string representing the full MIME type (maintype/subtype)
        • the string representing the maintype
        • the empty string

    If none of these keys produce a handler, raise a KeyError for the full MIME type.
set_content\((msg, obj, *args, **kw)\)

If the maintype is multipart, raise a TypeError; otherwise look up a handler function based on the type of obj (see next paragraph), call clear_content() on the msg, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store obj into msg, possibly making other changes to msg as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of obj (\(\text{typ} = \text{type}(\text{obj})\)), and look for the following keys in the registry, stopping with the first one found:

- the type itself (\(\text{typ}\))
- the type's fully qualified name (\(\text{typ}.__\text{module__} + \ '.' + \text{typ}.__\text{qualname__}\)).
- the type's qualified name (\(\text{typ}.__\text{qualname__}\))
- the type's name (\(\text{typ}.__\text{name__}\)).

If none of the above match, repeat all of the checks above for each of the types in the MRO (\(\text{typ}.__\text{mro__}\)). Finally, if no other key yields a handler, check for a handler for the key None. If there is no handler for None, raise a KeyError for the fully qualified name of the type.

Also add a MIME-Version header if one is not present (see also MIMEPart).

add_get_handler\((\text{key}, \text{handler})\)

Record the function handler as the handler for key. For the possible values of key, see get_content().
add_set_handler(typekey, handler)

Record handler as the function to call when an object of a type matching typekey is passed to set_content(). For the possible values of typekey, see set_content().
Content Manager Instances

Currently the email package provides only one concrete content manager, `raw_data_manager`, although more may be added in the future. `raw_data_manager` is the `content_manager` provided by `EmailPolicy` and its derivatives.

`email.contentmanager.raw_data_manager`

This content manager provides only a minimum interface beyond that provided by `Message` itself: it deals only with text, raw byte strings, and `Message` objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

`email.contentmanager.get_content(msg, errors='replace')`

Return the payload of the part as either a string (for text parts), an `EmailMessage` object (for `message/rfc822` parts), or a bytes object (for all other non-multipart types). Raise a `KeyError` if called on a multipart. If the part is a text part and `errors` is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

`email.contentmanager.set_content(msg, subtype="plain", charset='utf-8' cte=None, disposition=None,`
Add headers and payload to `msg`:

Add a **Content-Type** header with a maintype/subtype value.

- For **str**, set the MIME maintype to **text**, and set the subtype to **subtype** if it is specified, or **plain** if it is not.
- For **bytes**, use the specified maintype and subtype, or raise a **TypeError** if they are not specified.
- For **EmailMessage** objects, set the maintype to **message**, and set the subtype to **subtype** if it is specified or **rfc822** if it is not. If **subtype** is **partial**, raise an error (**bytes** objects must be used to construct **message/partial** parts).

If **charset** is provided (which is valid only for **str**), encode the string to bytes using the specified character set. The default is **utf-8**. If the specified **charset** is a known alias for a standard MIME charset name, use the standard charset instead.

If **cte** is set, encode the payload using the specified content transfer encoding, and set the **Content-Transfer-Encoding** header to that value. Possible values for **cte** are **quoted-printable**, **base64**, **7bit**, **8bit**, and **binary**. If the input cannot be encoded in the specified encoding (for example, specifying a **cte** of **7bit** for an input that contains non-ASCII
values), raise a `ValueError`.

- For `str` objects, if `cte` is not set use heuristics to determine the most compact encoding.
- For `EmailMessage`, per RFC 2046, raise an error if a `cte` of quoted-printable or base64 is requested for subtype rfc822, and for any `cte` other than 7bit for subtype external-body. For message/rfc822, use 8bit if `cte` is not specified. For all other values of `subtype`, use 7bit.

**Note:** A `cte` of binary does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but `BytesGenerator` does not serialize it correctly.

If `disposition` is set, use it as the value of the `Content-Disposition` header. If not specified, and `filename` is specified, add the header with the value `attachment`. If `disposition` is not specified and `filename` is also not specified, do not add the header. The only valid values for `disposition` are `attachment` and `inline`.

If `filename` is specified, use it as the value of the `filename` parameter of the `Content-Disposition` header.

If `cid` is specified, add a `Content-ID` header with `cid` as its value.

If `params` is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the `Content-Type` header.

If `headers` is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects
(distinguished from strings by having a `name` attribute), add the headers to `msg`.

**Footnotes**

[1] Originally added in 3.4 as a **provisional module**
email: Examples

Here are a few examples of how to use the email package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```python
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

    # me == the sender's email address
    # you == the recipient's email address
    msg['Subject'] = 'The contents of %s' % textfile
    msg['From'] = me
    msg['To'] = you

    # Send the message via our own SMTP server.
    s = smtplib.SMTP('localhost')
    s.send_message(msg)
    s.quit()
```

Parsing RFC 822 headers can easily be done by the using the classes
from the **parser** module:

```python
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

# Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr('From: Foo Bar <user@example.com>
To: <someone_else@example.com>
Subject: Test message

'Body would go here
')

# Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0]))
print('Sender name: {}'.format(headers['from'].addresses[0]))
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```python
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr
```
# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'Our family reunion'

# Open the files in binary mode. Use imghdr to figure out
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
        msg.add_attachment(img_data, maintype='image', subtype=imghdr.what)

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)

Here's an example of how to send the entire contents of a directory as an email message: [1]

#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes
from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP

def main():
    parser = ArgumentParser(description=""
    Send the contents of a directory as a MIME message. Unless the -o option is given, the email is sent by SMTP server, which then does the normal delivery process. Your local machine must be running an SMTP server.
    ""
    parser.add_argument('-d', '--directory',
                        help="Mail the contents of the specified directory, otherwise use the current directory. Only the regular files in the directory are sent, and we don't recurse to subdirectories.""
    )
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="Print the composed message to FILE instead of sending the message to the SMTP server."
    )
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)'
    )
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)'
    )

    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = 'Contents of directory %s' % os
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
msg.preamble = 'You will not see this in a MIME-aware mail reader.'

for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    if not os.path.isfile(path):
        continue
    # Guess the content type based on the file's extension.
    # will be ignored, although we should check gzip'd or compressed files.
    ctype, encoding = mimetypes.guess_type(path)
    if ctype is None or encoding is not None:
        # No guess could be made, or the file is encoded (compressed),
        # use a generic bag-of-bits type.
        ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)

if __name__ == '__main__':
    main()

Here's an example of how to unpack a MIME message like the one above, into a directory of files:

#!/usr/bin/env python3
import os
import email
import mimetypes

from email.policy import default
from argparse import ArgumentParser

def main():
    parser = ArgumentParser(description="""Unpack a MIME message into a directory of files.""
    parser.add_argument('-d', '--directory', required=True,
        help="""Unpack the MIME message into the named directory, which will be created if it doesn't already exist."""
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given
        # filenames before changing them.

    counter = 0
Here’s an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```python
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com"
msg['To'] = (Address("Penelope Pussycat", "penelope"
                Address("Fabrette Pussycat", "fabrette"
msg.set_content(""")
```
Salut!


--Pepé
""

# Add the html version. This converts the message into a multipart/alternative container, with the original text message as the first part and the new html message as the second part.

asparagus_cid = make_msgid()
msg.add_alternative("""\n<html>
<head></head>
<body>
  <p>Salut!</p>
  <p>Cela ressemble à un excellent recipie</p>
</body>
</html>"").format(asparagus_cid=asparagus_cid[1:-1]), subtype

# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.

with open("roasted-asparagus.jpg", 'rb') as img:
  msg.get_payload()[1].add_related(img.read(), 'image',
                       cid=asparagus_cid)

# Make a local copy of what we are going to send.

with open('outgoing.msg', 'wb') as f:
  f.write(bytes(msg))
# Send the message via local SMTP server.

```python
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```python
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, the least formatted payload is and print the first three lines. If the message has no plain text part printing the preview is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
```
print()
print(''.join(simplest.get_content().splitlines(keepends=True)))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to find the full message:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
    sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())
        else:
            extension = mimetypes.guess_extension(part.get_filename())
        with tempfile.NamedTemporaryFile(suffix=extension) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form to HTML form.
            partfiles[part['content-id'][1:-1]] = f
        else:
            print("Don't know how to display {}".format(richest))
            sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href=
```python
# point to the filenames in partfiles. It also # of the html. It could be written using html.parser.
f.write(magic_html_parser(body.get_content(), partfiles.values))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
os.remove(fn)
```

Up to the prompt, the output from the above is:

To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!


---

**Footnotes**

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.
**email.message.Message:** Representing an email message using the **compat32** API

The **Message** class is very similar to the **EmailMessage** class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the **EmailMessage** class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for **Message**) policy **Compat32**. If you are going to use another policy, you should be using the **EmailMessage** class instead.

An email message consists of **headers** and a **payload**. Headers must be **RFC 5233** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as **multipart/** or **message/rfc822**.

The conceptual model provided by a **Message** object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the
payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The **Message** pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the *From_* header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/* and *message/rfc822*).

Here are the methods of the **Message** class:

```python
class email.message.Message(policy=compat32)
    If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

    *Changed in version 3.3:* The *policy* keyword argument was added.

    `as_string(unixfrom=False, maxheaderlen=0, policy=None)`
    Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to False. For backward compatibility reasons, *maxheaderlen* defaults to 0, so if you want a different
value you must override it explicitly (the value specified for `max_line_length` in the policy will be ignored by this method). The `policy` argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified `policy` will be passed to the `Generator`.

Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a `Generator` instance and use its `flatten()` method directly. For example:

```python
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=50)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode “unknown character” code points. (See also `as_bytes()` and `BytesGenerator`.)

*Changed in version 3.4:* the `policy` keyword argument was added.
__str__()

Equivalent to `as_string()`. Allows `str(msg)` to produce a string containing the formatted message.

`as_bytes(unixfrom=False, policy=None)`

Return the entire message flattened as a bytes object. When optional `unixfrom` is true, the envelope header is included in the returned string. `unixfrom` defaults to `False`. The `policy` argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified `policy` will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a `BytesGenerator` instance and use its `flatten()` method directly. For example:

```python
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen)
g.flatten(msg)
text = fp.getvalue()
```
New in version 3.4.

```
__bytes__()
```
Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

New in version 3.4.

```
is_multipart()
```
Return `True` if the message's payload is a list of sub-`Message` objects, otherwise return `False`. When `is_multipart()` returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that `is_multipart()` returning `True` does not necessarily mean that “msg.get_content_maintype() == ‘multipart’” will return the `True`. For example, `is_multipart` will return `True` when the `Message` is of type `message/rfc822`.)

```
set_unixfrom(unixfrom)
```
Set the message’s envelope header to `unixfrom`, which should be a string.

```
get_unixfrom()
```
Return the message’s envelope header. Defaults to `None` if the envelope header was never set.

```
attach(payload)
```
Add the given `payload` to the current payload, which must be `None` or a list of `Message` objects before the call. After the call, the payload will always be a list of `Message` objects. If you want to set the payload to a scalar object (e.g. a string), use `set_payload()` instead.
This is a legacy method. On the EmailMessage class its functionality is replaced by `set_content()` and the related `make` and `add` methods.

**get_payload**(*i=None, decode=False*)

Return the current payload, which will be a list of `Message` objects when `is_multipart()` is `True`, or a string when `is_multipart()` is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument `i`, `get_payload()` will return the `i`-th element of the payload, counting from zero, if `is_multipart()` is `True`. An `IndexError` will be raised if `i` is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. `is_multipart()` is `False`) and `i` is given, a `TypeError` is raised.

Optional `decode` is a flag indicating whether the payload should be decoded or not, according to the `Content-Transfer-Encoding` header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or `Content-Transfer-Encoding` header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the `decode` flag is `True`, then None is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (InvalidBase64PaddingDefect or
InvalidBase64CharactersDefect, respectively).

When \textit{decode} is False (the default) the body is returned as a string without decoding the \textit{Content-Transfer-Encoding}. However, for a \textit{Content-Transfer-Encoding} of 8bit, an attempt is made to decode the original bytes using the \texttt{charset} specified by the \textit{Content-Type} header, using the \texttt{replace} error handler. If no \texttt{charset} is specified, or if the \texttt{charset} given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the \texttt{EmailMessage} class its functionality is replaced by \texttt{get_content()} and \texttt{iter_parts()}.

\texttt{set\_payload}(\textit{payload}, \texttt{charset}=	exttt{None})

Set the entire message object's payload to \textit{payload}. It is the client's responsibility to ensure the payload invariants. Optional \texttt{charset} sets the message's default character set; see \texttt{set\_charset()} for details.

This is a legacy method. On the \texttt{EmailMessage} class its functionality is replaced by \texttt{set\_content()}.

\texttt{set\_charset}(\textit{charset})

Set the character set of the payload to \textit{charset}, which can either be a \texttt{Charset} instance (see \texttt{email.charset}), a string naming a character set, or \texttt{None}. If it is a string, it will be converted to a \texttt{Charset} instance. If \texttt{charset} is \texttt{None}, the \texttt{charset} parameter will be removed from the \textit{Content-Type} header (the message will not be otherwise modified). Anything else will generate a \texttt{TypeError}. 
If there is no existing `MIME-Version` header one will be added. If there is no existing `Content-Type` header, one will be added with a value of `text/plain`. Whether the `Content-Type` header already exists or not, its `charset` parameter will be set to `charset.output_charset`. If `charset.input_charset` and `charset.output_charset` differ, the payload will be re-encoded to the `output_charset`. If there is no existing `Content-Transfer-Encoding` header, then the payload will be transfer-encoded, if needed, using the specified `Charset`, and a header with the appropriate value will be added. If a `Content-Transfer-Encoding` header already exists, the payload is assumed to already be correctly encoded using that `Content-Transfer-Encoding` and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `charset` parameter of the `email.emailmessage.EmailMessage.set_content()` method.

**get_charset()**

Return the `Charset` instance associated with the message's payload.

This is a legacy method. On the `EmailMessage` class it always returns `None`.

The following methods implement a mapping-like interface for accessing the message's `RFC 2822` headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed
order to the keys returned by `keys()`, but in a `Message` object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as `Header` objects with a charset of `unknown-8bit`.

`__len__()`  
Return the total number of headers, including duplicates.

`__contains__`(name)  
Return true if the message object has a field named `name`. Matching is done case-insensitively and `name` should not include the trailing colon. Used for the `in` operator, e.g.:

```python
if 'message-id' in myMessage:
    print('Message-ID:', myMessage['message-id'])
```

`__getitem__`(name)  
Return the value of the named header field. `name` should not include the colon field separator. If the header is missing, `None` is returned; a `KeyError` is never raised.
Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the `get_all()` method to get the values of all the extant named headers.

```python
__setitem__(name, val)
```

Add a header to the message with field name `name` and value `val`. The field is appended to the end of the message's existing fields.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name `name`, delete the field first, e.g.:

```python
del msg['subject']
msg['subject'] = 'Python roolz!'
```

```python
__delitem__(name)
```

Delete all occurrences of the field with name `name` from the message's headers. No exception is raised if the named field isn't present in the headers.

```python
keys()
```

Return a list of all the message's header field names.

```python
values()
```

Return a list of all the message's field values.

```python
items()
```

Return a list of 2-tuples containing all the message's field headers and values.
get(name, failobj=None)

Return the value of the named header field. This is identical to __getitem__() except that optional failobj is returned if the named header is missing (defaults to None).

Here are some additional useful methods:

get_all(name, failobj=None)

Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).

add_header(_name, _value, **_params)

Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as key="value" unless the value is None, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see RFC 2231 for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in RFC 2231 format using a CHARSET of utf-8 and a
Here's an example:

```python
msg.add_header('Content-Disposition', 'attachment')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```python
msg.add_header('Content-Disposition', 'attachment', filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

**replace_header**(_name,_value)

Replace a header. Replace the first header found in the message that matches _name, retaining header order and field name case. If no matching header was found, a KeyError is raised.

**get_content_type()**

Return the message's content type. The returned string is coerced to lower case of the form maintype/subtype. If there was no Content-Type header in the message the default type as given by get_default_type() will be returned. Since according to RFC 2045, messages always have a default type,
get_content_type() will always return a value.

RFC 2045 defines a message’s default type to be text/plain unless it appears inside a multipart/digest container, in which case it would be message/rfc822. If the Content-Type header has an invalid type specification, RFC 2045 mandates that the default type be text/plain.

get_content_maintype()
Return the message’s main content type. This is the maintype part of the string returned by get_content_type().

get_content_subtype()
Return the message’s sub-content type. This is the subtype part of the string returned by get_content_type().

get_default_type()
Return the default content type. Most messages have a default content type of text/plain, except for messages that are subparts of multipart/digest containers. Such subparts have a default content type of message/rfc822.

set_default_type(ctype)
Set the default content type. ctype should either be text/plain or message/rfc822, although this is not enforced. The default content type is not stored in the Content-Type header.

get_params(failobj=None, header='content-type', unquote=True)
Return the message's Content-Type parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the '=' sign. The left hand side of the '=' is the key, while the right hand side is the value. If there is no '=' sign in
the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional `unquote` is `True` (the default).

Optional `failobj` is the object to return if there is no `Content-Type` header. Optional `header` is the header to search instead of `Content-Type`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `params` property of the individual header objects returned by the header access methods.

```python
get_param(param, failobj=None, header='content-type', unquote=True)
```

Return the value of the `Content-Type` header's parameter `param` as a string. If the message has no `Content-Type` header or if there is no such parameter, then `failobj` is returned (defaults to `None`).

Optional `header` if given, specifies the message header to use instead of `Content-Type`.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was RFC 2231 encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn’t care whether the parameter was encoded as in RFC 2231, you can collapse the parameter value by calling `email.utils.collapse_rfc2231_value()`,
passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn’t. For example:

```python
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless `unquote` is set to `False`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `params` property of the individual header objects returned by the header access methods.

```python
set_param(param, value, header='Content-Type', requote=True, charset=None, language='', replace=False)
```

Set a parameter in the `Content-Type` header. If the parameter already exists in the header, its value will be replaced with `value`. If the `Content-Type` header as not yet been defined for this message, it will be set to `text/plain` and the new parameter value will be appended as per RFC 2045.

Optional `header` specifies an alternative header to `Content-Type`, and all parameters will be quoted as necessary unless optional `requote` is `False` (the default is `True`).

If optional `charset` is specified, the parameter will be encoded according to RFC 2231. Optional `language` specifies the RFC 2231 language, defaulting to the empty string. Both `charset` and `language` should be strings.
If `replace` is `False` (the default) the header is moved to the end of the list of headers. If `replace` is `True`, the header will be updated in place.

*Changed in version 3.4:* replace keyword was added.

**del_param**

```python
def del_param(param, header='content-type', requote=True)
```

Remove the given parameter completely from the `Content-Type` header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless `requote` is `False` (the default is `True`). Optional `header` specifies an alternative to `Content-Type`.

**set_type**

```python
def set_type(type, header='Content-Type', requote=True)
```

Set the main type and subtype for the `Content-Type` header. `type` must be a string in the form `maintype/subtype`, otherwise a `ValueError` is raised.

This method replaces the `Content-Type` header, keeping all the parameters in place. If `requote` is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the `header` argument. When the `Content-Type` header is set a `MIME-Version` header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

**get_filename**

```python
def get_filename(failobj=None)
```

Return the value of the `filename` parameter of the `Content-Disposition` header of the message. If the header does not have
a filename parameter, this method falls back to looking for the name parameter on the Content-Type header. If neither is found, or the header is missing, then failobj is returned. The returned string will always be unquoted as per email.utils.unquote().

**get_boundary**\(\text{(failobj=\text{None})}\)

Return the value of the boundary parameter of the Content-Type header of the message, or failobj if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per email.utils.unquote().

**set_boundary**\(\text{(boundary)}\)

Set the boundary parameter of the Content-Type header to boundary. set_boundary() will always quote boundary if necessary. A HeaderParseError is raised if the message object has no Content-Type header.

Note that using this method is subtly different than deleting the old Content-Type header and adding a new one with the new boundary via add_header(), because set_boundary() preserves the order of the Content-Type header in the list of headers. However, it does not preserve any continuation lines which may have been present in the original Content-Type header.

**get_content_charset**\(\text{(failobj=\text{None})}\)

Return the charset parameter of the Content-Type header, coerced to lower case. If there is no Content-Type header, or if that header has no charset parameter, failobj is returned.

Note that this method differs from get_charset() which
returns the `Charset` instance for the default encoding of the message body.

**get_charsets** *(failobj=None)*

Return a list containing the character set names in the message. If the message is a `multipart`, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the `Content-Type` header for the represented subpart. However, if the subpart has no `Content-Type` header, no `charset` parameter, or is not of the `text` main MIME type, then that item in the returned list will be `failobj`.

**get_content_disposition**

Return the lowercased value (without parameters) of the message’s `Content-Disposition` header if it has one, or `None`. The possible values for this method are `inline`, `attachment` or `None` if the message follows RFC 2183.

*New in version 3.5.*

**walk()**

The `walk()` method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use `walk()` as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:
walk iterates over the subparts of any part where is_multipart() returns True, even though msg.get_content_maintype() == 'multipart' may return False. We can see this in our example by making use of the _structure debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart'
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False True
False False
```
Here the message parts are not multipart, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

**Message** objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

**preamble**

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message’s *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

**epilogue**

The *epilogue* attribute acts the same way as the *preamble*
attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the Generator to print a newline at the end of the file.

**defects**

The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.
email.mime: Creating email and MIME objects from scratch

Source code: Lib/email/mime/

This module is part of the legacy (Compat32) email API. Its functionality is partially replaced by the contentmanager in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual Message objects by hand. In fact, you can also take an existing structure and add new Message objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating Message instances, adding attachments and all the appropriate headers manually. For MIME messages though, the email package provides some convenient subclasses to make things easier.

Here are the classes:

```python
class email.mime.base.MIMEBase(_maintype, _subtype, *,
    policy=compat32, **_params)
```

Module: email.mime.base

This is the base class for all the MIME-specific subclasses of
**Message.** Ordinarily you won’t create instances specifically of `MIMEBase`, although you could. `MIMEBase` is provided primarily as a convenient base class for more specific MIME-aware subclasses.

`_maintype` is the `Content-Type` major type (e.g. `text` or `image`), and `_subtype` is the `Content-Type` minor type (e.g. `plain` or `gif`). `_params` is a parameter key/value dictionary and is passed directly to `Message.add_header`.

If `policy` is specified, (defaults to the `compat32` policy) it will be passed to `Message`.

The `MIMEBase` class always adds a `Content-Type` header (based on `_maintype`, `_subtype`, and `_params`), and a `MIME-Version` header (always set to 1.0).

*Changed in version 3.6:* Added `policy` keyword-only parameter.

class email.mime.nonmultipart.*MIMENonMultipart*

Module: email.mime.nonmultipart

A subclass of `MIMEBase`, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the `attach()` method, which only makes sense for *multipart* messages. If `attach()` is called, a `MultipartConversionError` exception is raised.

class email.mime.multipart.*MIMEMultipart*( _subtype='mixed', boundary=None, _subparts=None, *, policy=compat32, **_params )

Module: email.mime.multipart

A subclass of `MIMEBase`, this is an intermediate base class for
MIME messages that are *multipart*. Optional `_subtype` defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

*_subparts* is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the `Message.attach` method.

Optional *policy* argument defaults to *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the `_params` argument, which is a keyword dictionary.

*Changed in version 3.6:* Added *policy* keyword-only parameter.

```python
class email.mime.application.MIMEApplication(_data, 
    _subtype='octet-stream', _encoder=email.encoders.encode_base64, *, 
    policy=compat32, **_params)
    Module: email.mime.application
```

A subclass of `MIMENonMultipart`, the `MIMEApplication` class is used to represent MIME message objects of major type *application*. `_data` is a string containing the raw byte data. Optional `_subtype` specifies the MIME subtype and defaults to *octet-stream*.

Optional `_encoder` is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one
argument, which is the MIMEApplication instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

Optional policy argument defaults to compat32.

_params are passed straight through to the base class constructor.

Changed in version 3.6: Added policy keyword-only parameter.

class email.mime.audio.MIMEAudio(_audiodata, subtype=None, encoder=email.encoders.encode_base64, *, policy=compat32, **_params)

Module: email.mime.audio

A subclass of MIMENonMultipart, the MIMEAudio class is used to create MIME message objects of major type audio. _audiodata is a string containing the raw audio data. If this data can be decoded by the standard Python module sndhdr, then the subtype will be automatically included in the Content-Type header. Otherwise you can explicitly specify the audio subtype via the _subtype argument. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the MIMEAudio instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default
encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional `policy` argument defaults to `compat32`.

`_params` are passed straight through to the base class constructor.

*Changed in version 3.6:* Added `policy` keyword-only parameter.

```python
class email.mime.image.MIMEImage(_imagedata, _subtype=None, _encoder=email.encoders.encode_base64, *, policy=compat32, **_params)
```

Module: `email.mime.image`

A subclass of `MIMENonMultipart`, the `MIMEImage` class is used to create MIME message objects of major type `image`. `_imagedata` is a string containing the raw image data. If this data can be decoded by the standard Python module `imghdr`, then the subtype will be automatically included in the `Content-Type` header. Otherwise you can explicitly specify the image subtype via the `_subtype` argument. If the minor type could not be guessed and `_subtype` was not given, then `TypeError` is raised.

Optional `_encoder` is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any `Content-Transfer-Encoding` or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional `policy` argument defaults to `compat32`. 
_params are passed straight through to the MIMEBase constructor.

*Changed in version 3.6:* Added policy keyword-only parameter.

class email.mime.message.MIMEMessage(_msg,  
_subtype='rfc822', *, policy=compat32)

  Module: email.mime.message

  A subclass of MIMENonMultipart, the MIMEMessage class is used to create MIME objects of main type message. _msg is used as the payload, and must be an instance of class Message (or a subclass thereof), otherwise a TypeError is raised.

  Optional _subtype sets the subtype of the message; it defaults to rfc822.

  Optional policy argument defaults to compat32.

*Changed in version 3.6:* Added policy keyword-only parameter.

class email.mime.text.MIMEText(_text, _subtype='plain',  
_charset=None, *, policy=compat32)

  Module: email.mime.text

  A subclass of MIMENonMultipart, the MIMEText class is used to create MIME objects of major type text. _text is the string for the payload. _subtype is the minor type and defaults to plain. _charset is the character set of the text and is passed as an argument to the MIMENonMultipart constructor; it defaults to us-ascii if the string contains only ascii code points, and utf-8 otherwise. The _charset parameter accepts either a string or a Charset instance.

  Unless the _charset argument is explicitly set to None, the
MIMEText object created will have both a `Content-Type` header with a `charset` parameter, and a `Content-Transfer-Encoding` header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a charset is passed in the `set_payload` command. You can "reset" this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new `Content-Transfer-Encoding` header).

Optional `policy` argument defaults to `compat32`.

*Changed in version 3.5:* `_charset` also accepts `Charset` instances.

*Changed in version 3.6:* Added `policy` keyword-only parameter.
**email.header**: Internationalized headers

**Source code**: Lib/email/header.py

This module is part of the legacy (Compat32) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the `EmailMessage` class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

**RFC 2822** is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The `email` package supports these standards in its `email.header` and `email.charset` modules.
If you want to include non-ASCII characters in your email headers, say
in the Subject or To fields, you should use the Header class and
assign the field in the Message object to an instance of Header
instead of using a string for the header value. Import the Header class
from the email.header module. For example:
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
Notice here how we wanted the Subject field to contain a non-ASCII
character? We did this by creating a Header instance and passing in
the character set that the byte string was encoded in. When the
subsequent Message instance was flattened, the Subject field was
properly RFC 2047 encoded. MIME-aware mail readers would show
this header using the embedded ISO-8859-1 character.
Here is the Header class description:
class email.header. Header(s=None, charset=None,
maxlinelen=None, header_name=None, continuation_ws=' ',
errors='strict')
Create a MIME-compliant header that can contain strings in
different character sets.
Optional s is the initial header value. If None (the default), the initial
header value is not set. You can later append to the header with
append() method calls. s may be an instance of bytes or str,
but see the append() documentation for semantics.


Optional `charset` serves two purposes: it has the same meaning as the `charset` argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the `charset` argument. If `charset` is not provided in the constructor (the default), the `us-ascii` character set is used both as `s`'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicitly via `maxlinelen`. For splitting the first line to a shorter value (to account for the field header which isn't included in `s`, e.g. `Subject`) pass in the name of the field in `header_name`. The default `maxlinelen` is 76, and the default value for `header_name` is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional `continuation_ws` must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. `continuation_ws` defaults to a single space character.

Optional `errors` is passed straight through to the `append()` method.

```python
append(s, charset=None, errors='strict')
```

Append the string `s` to the MIME header.

Optional `charset`, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the `charset` given in the constructor is used.

`s` may be an instance of `bytes` or `str`. If it is an instance of `bytes`, then `charset` is the encoding of that byte string, and a
UnicodeError will be raised if the string cannot be decoded with that character set.

If s is an instance of str, then charset is a hint specifying the character set of the characters in the string.

In either case, when producing an RFC 2822-compliant header using RFC 2047 rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional errors is passed as the errors argument to the decode call if s is a byte string.

encode(splitchars='; ', maxlinelen=None, linesep='\n')
Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional splitchars is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of RFC 2822's 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect RFC 2047 encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.
linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (\n), but \r\n can be specified in order to produce headers with RFC-compliant line separators.

*Changed in version 3.2:* Added the linesep argument.

The `Header` class also provides a number of methods to support standard operators and built-in functions.

```
__str__(self)
```

Returns an approximation of the `Header` as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of 'unknown-8bit' are decoded as ASCII using the 'replace' error handler.

*Changed in version 3.2:* Added handling for the 'unknown-8bit' charset.

```
__eq__(other)
```

This method allows you to compare two `Header` instances for equality.

```
__ne__(other)
```

This method allows you to compare two `Header` instances for inequality.

The `email.header` module also provides the following convenient functions.

```
email.header.decode_header(header)
```

Decode a message header value without converting the character
set. The header value is in header.
This function returns a list of (decoded_string, charset)
pairs containing each of the decoded parts of the header. charset is
None for non-encoded parts of the header, otherwise a lower case
string containing the name of the character set specified in the
encoded string.
Here’s an example:
>>> from email.header import decode_header
[(b'p\xf6stal', 'iso-8859-1')]
email.header. make_header(decoded_seq, maxlinelen=None,
header_name=None, continuation_ws=' ')
Create a Header instance from a sequence of pairs as returned by
decode_header().
decode_header() takes a header value string and returns a
sequence of pairs of the format (decoded_string, charset)
where charset is the name of the character set.
This function takes one of those sequence of pairs and returns a
Header instance. Optional maxlinelen, header_name, and
continuation_ws are as in the Header constructor.
Python » 3.7.3 Documentation » The Python Standard

previous | next | modules | index

Library » Internet Data Handling » email — An email and MIME handling package »
© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.


email.charset: Representing character sets

Source code: Lib/email/charset.py

This module is part of the legacy (Compat32) email API. In the new API only the aliases table is used.

The remaining text in this section is the original documentation of the module.

This module provides a class Charset for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of Charset are used in several other modules within the email package.

Import this class from the email.charset module.

class email.charset.Charset(input_charset=DEFAULT_CHARSET)
    Map character sets to their email properties.

    This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.
Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional input_charset is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if input_charset is iso-8859-1, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If input_charset is euc-jp, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the euc-jp character set to the iso-2022-jp character set.

Charset instances have the following data attributes:

**input_charset**
The initial character set specified. Common aliases are converted to their official email names (e.g. latin_1 is converted to iso-8859-1). Defaults to 7-bit us-ascii.

**header_encoding**
If the character set must be encoded before it can be used in an email header, this attribute will be set to Charset.QP (for quoted-printable), Charset.BASE64 (for base64 encoding), or Charset.SHORTEST for the shortest of QP or BASE64 encoding. Otherwise, it will be None.

**body_encoding**
Same as header_encoding, but describes the encoding for the mail message's body, which indeed may be different than the
header encoding. \texttt{Charset.SHORTEST} is not allowed for \texttt{body_encoding}.

\textbf{output\_charset}

Some character sets must be converted before they can be used in email headers or bodies. If the \texttt{input\_charset} is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be \texttt{None}.

\textbf{input\_codec}

The name of the Python codec used to convert the \texttt{input\_charset} to Unicode. If no conversion codec is necessary, this attribute will be \texttt{None}.

\textbf{output\_codec}

The name of the Python codec used to convert Unicode to the \texttt{output\_charset}. If no conversion codec is necessary, this attribute will have the same value as the \texttt{input\_codec}.

\texttt{Charset} instances also have the following methods:

\textbf{get\_body\_encoding()}

Return the content transfer encoding used for body encoding.

This is either the string \texttt{quoted-printable} or \texttt{base64} depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the \texttt{Content-Transfer-Encoding} header itself to whatever is appropriate.

Returns the string \texttt{quoted-printable} if \texttt{body\_encoding} is \texttt{QP}, returns the string \texttt{base64} if \texttt{body\_encoding} is \texttt{BASE64},
and returns the string 7bit otherwise.

**get_output_charset()**
Return the output character set.

This is the output_charset attribute if that is not None, otherwise it is input_charset.

**header_encode(string)**
Header-encode the string string.

The type of encoding (base64 or quoted-printable) will be based on the header_encoding attribute.

**header_encode_lines(string, maxlengths)**
Header-encode a string by converting it first to bytes.

This is similar to header_encode() except that the string is fit into maximum line lengths as given by the argument maxlengths, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

**body_encode(string)**
Body-encode the string string.

The type of encoding (base64 or quoted-printable) will be based on the body_encoding attribute.

The CharSet class also provides a number of methods to support standard operations and built-in functions.

**__str__()**
Returns input_charset as a string coerced to lower case. __repr__() is an alias for __str__().
__eq__(other)
This method allows you to compare two Charset instances for equality.

__ne__(other)
This method allows you to compare two Charset instances for inequality.

The email.charset module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.add_charset(charset, header_enc=None, body_enc=None, output_charset=None)
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional header_enc and body_enc is either Charset.QP for quoted-printable, Charset.BASE64 for base64 encoding, Charset.SHORTEST for the shortest of quoted-printable or base64 encoding, or None for no encoding. SHORTEST is only valid for header_enc. The default is None for no encoding.

Optional output_charset is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method Charset.convert() is called. The default is to output in the same character set as the input.

Both input_charset and output_charset must have Unicode codec entries in the module's character set-to-codec mapping; use
add_codec() to add codecs the module does not know about. See the codecs module's documentation for more information.

The global character set registry is kept in the module global dictionary CHARSETS.

email.charset.add_alias(alias, canonical)
Add a character set alias. alias is the alias name, e.g. latin-1. canonical is the character set's canonical name, e.g. iso-8859-1.

The global charset alias registry is kept in the module global dictionary ALIASES.

eemail.charset.add_codec(charset, codecname)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. codecname is the name of a Python codec, as appropriate for the second argument to the str's encode() method.
email.encoders: Encoders

Source code: Lib/email/encoders.py

This module is part of the legacy (Compat32) email API. In the new API the functionality is provided by the cte parameter of the set_content() method.

The remaining text in this section is the original documentation of the module.

When creating Message objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for image/* and text/* type messages containing binary data.

The email package provides some convenient encodings in its encoders module. These encoders are actually used by the MIMEAudio and MIMEImage class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the Content-Transfer-Encoding header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a TypeError if passed a message whose type is multipart.

Here are the encoding functions provided:

email.encoders.encode_quopri(msg)
Encodes the payload into quoted-printable form and sets the `Content-Transfer-Encoding` header to quoted-printable [1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

**email.encoders.**`encode_base64(msg)`

Encodes the payload into base64 form and sets the `Content-Transfer-Encoding` header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

**email.encoders.**`encode_7or8bit(msg)`

This doesn’t actually modify the message’s payload, but it does set the `Content-Transfer-Encoding` header to either 7bit or 8bit as appropriate, based on the payload data.

**email.encoders.**`encode_noop(msg)`

This does nothing; it doesn’t even set the `Content-Transfer-Encoding` header.

**Footnotes**

[1] Note that encoding with `encode_quopri()` also encodes all tabs and space characters in the data.
email — An email and MIME handling package
There are a couple of useful utilities provided in the `email.utils` module:

**email.utils.localtime** *(dt=None)*

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a `datetime` instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, *dt.tzinfo* is `None`), it is assumed to be in local time. In this case, a positive or zero value for *isdst* causes `localtime` to presume initially that summer time (for example, Daylight Saving Time) is or is not (respectively) in effect for the specified time. A negative value for *isdst* causes the `localtime` to attempt to divine whether summer time is in effect for the specified time.

*New in version 3.3.*

**email.utils.make_msgid** *(idstring=None, domain=None)*

Returns a string suitable for an RFC 2822-compliant `Message-ID` header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the ‘@’. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed
system that uses a consistent domain name across multiple hosts.

*Changed in version 3.2:* Added the *domain* keyword.

The remaining functions are part of the legacy (Compat32) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

**email.utils.**`quote`(str)

Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

**email.utils.**`unquote`(str)

Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

**email.utils.**`parseaddr`(address)

Parse address – which should be the value of some address-containing field such as *To* or *Cc* – into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of (',', '') is returned.

**email.utils.**`formataddr`(pair, charset='utf-8')

The inverse of `parseaddr()`, this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

Optional *charset* is the character set that will be used in the RFC 2047 encoding of the *realname* if the *realname* contains non-
ASCII characters. Can be an instance of `str` or a `Charset`. Defaults to `utf-8`.

*Changed in version 3.3:* Added the `charset` option.

e-mail.utils.**getaddresses**(*fieldvalues*)

This method returns a list of 2-tuples of the form returned by `parseaddr()`. `fieldvalues` is a sequence of header field values as might be returned by `Message.get_all`. Here’s a simple example that gets all the recipients of a message:

```python
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

e-mail.utils.**parsedate**(*date*)

Attempts to parse a date according to the rules in [RFC 2822](https://tools.ietf.org/html/rfc2822). However, some mailers don’t follow that format as specified, so `parsedate()` tries to guess correctly in such cases. `date` is a string containing an [RFC 2822](https://tools.ietf.org/html/rfc2822) date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, `parsedate()` returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

e-mail.utils.**parsedate_tz**(*date*)

Performs the same function as `parsedate()`, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can
be passed directly to `time.mktime()`, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time) [1]. If the input string has no timezone, the last element of the tuple returned is `None`. Note that indexes 6, 7, and 8 of the result tuple are not usable.

```python
email.utils.parsedate_to_datetime(date)
```

The inverse of `format_datetime()`. Performs the same function as `parsedate()`, but on success returns a `datetime`. If the input date has a timezone of `-0000`, the datetime will be a naive `datetime`, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the `datetime` will be an aware `datetime` with the corresponding a `timezone tzinfo`.

*New in version 3.3.*

```python
email.utils.mktime_tz(tuple)
```

Turn a 10-tuple as returned by `parsedate_tz()` into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

```python
email.utils.formatdate(timeval=None, localtime=False, usegmt=False)
```

Returns a date string as per RFC 2822, e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional `timeval` if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.
Optional *localtime* is a flag that when *True*, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is *False* meaning UTC is used.

Optional *usegmt* is a flag that when *True*, outputs a date string with the timezone as an ascii string *GMT*, rather than a numeric *-0000*. This is needed for some protocols (such as HTTP). This only applies when *localtime* is *False*. The default is *False*.

```
email.utils.format_datetime(dt, usegmt=False)
```
Like *formatdate*, but the input is a *datetime* instance. If it is a naive datetime, it is assumed to be “UTC with no information about the source timezone”, and the conventional *-0000* is used for the timezone. If it is an aware *datetime*, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then *usegmt* may be set to *True*, in which case the string *GMT* is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

*New in version 3.3.*

```
email.utils.decode_rfc2231(s)
```
Decode the string *s* according to *RFC 2231*.

```
email.utils.encode_rfc2231(s, charset=None, language=None)
```
Encode the string *s* according to *RFC 2231*. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*. 
When a header parameter is encoded in **RFC 2231** format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional `errors` is passed to the `errors` argument of `str`'s `encode()` method; it defaults to `'replace'`. Optional `fallback_charset` specifies the character set to use if the one in the **RFC 2231** header is not known by Python; it defaults to `'us-ascii'`.

For convenience, if the `value` passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

**email.utils.decode_params**(params)

Decode parameters list according to **RFC 2231**. `params` is a sequence of 2-tuples containing elements of the form `(content-type,string-value)`.

**Footnotes**

[1] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822**.
email — An email and MIME handling package
email.iterators: Iterators

Source code: Lib/email/iterators.py

Iterating over a message object tree is fairly easy with the Message.walk method. The email.iterators module provides some useful higher level iterations over message object trees.

email.iterators.body_line_iterator(msg, decode=False)

This iterates over all the payloads in all the subparts of msg, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn’t a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using readline(), skipping over all the intervening headers.

Optional decode is passed through to Message.get_payload.

email.iterators.typed_subpart_iterator(msg, maintype='text', subtype=None)

This iterates over all the subparts of msg, returning only those subparts that match the MIME type specified by maintype and subtype.

Note that subtype is optional; if omitted, then subpart MIME type matching is done only with the main type. maintype is optional too; it defaults to text.

Thus, by default typed_subpart_iterator() returns each subpart that has a MIME type of text/*.
The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

`email.iterators._structure(msg, fp=None, level=0, include_default=False)`

Prints an indented representation of the content types of the message object structure. For example:

```python
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
text/plain
text/plain
multipart/digest
  message/rfc822
text/plain
  message/rfc822
text/plain
  message/rfc822
text/plain
  message/rfc822
text/plain
text/plain
```

Optional `fp` is a file-like object to print the output to. It must be suitable for Python's `print()` function. `level` is used internally. `include_default`, if true, prints the default type as well.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
**json — JSON encoder and decoder**

**Source code:** Lib/json/__init__.py

JSON (JavaScript Object Notation), specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript [1]).

`json` exposes an API familiar to users of the standard library `marshal` and `pickle` modules.

Encoding basic Python object hierarchies:

```python
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2), '["foo", {"bar": ["baz", null, 1.0, 2]]}'
>>> print(json.dumps("\\foo\bar"))
"\\foo\\bar"
>>> print(json.dumps('\\u1234'))
\\u1234
>>> print(json.dumps(''))
\\
>>> print(json.dumps({'c': 0, 'b': 0, 'a': 0}, sort_keys=True)
{'a': 0, 'b': 0, 'c': 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
['"streaming API"]
```
Compact encoding:
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators
'[1,2,3,{"4":5,"6":7}]'

Pretty printing:
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True
{
"4": 5,
"6": 7
}

Decoding JSON:
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]'
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']

Specializing JSON object decoding:
>>> import json
>>> def as_complex(dct):
...
if '__complex__' in dct:
...
return complex(dct['real'], dct['imag'])
...
return dct


Extending `JSONEncoder`:  

```python
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise
...         return json.JSONEncoder.default(self, obj)

>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'  
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'  
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ',', '1.0', ',', ']
```

Using `json.tool` from the shell to validate and pretty-print:

```bash
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: {'1.2:3.4'}
```
See Command Line Interface for detailed documentation.

**Note:** JSON is a subset of YAML 1.2. The JSON produced by this module’s default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.
Basic Usage

```python
json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

Serialize `obj` as a JSON formatted stream to `fp` (a supporting file-like object) using this conversion table.

If `skipkeys` is true (default: False), then dict keys that are not of a basic type (`str, int, float, bool, None`) will be skipped instead of raising a `TypeError`.

The `json` module always produces `str` objects, not `bytes` objects. Therefore, `fp.write()` must support `str` input.

If `ensure_ascii` is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `ensure_ascii` is false, these characters will be output as-is.

If `check_circular` is false (default: True), then the circular reference check for container types will be skipped and a circular reference will result in an `OverflowError` (or worse).

If `allow_nan` is false (default: True), then it will be a `ValueError` to serialize out of range `float` values (`nan, inf, -inf`) in strict compliance of the JSON specification. If `allow_nan` is true, their JavaScript equivalents (`NaN, Infinity, -Infinity`) will be used.

If `indent` is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `' '` will only insert newlines.
None (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If \( \text{indent} \) is a string (such as \"\t\"), that string is used to indent each level.

\textit{Changed in version 3.2:} Allow strings for \( \text{indent} \) in addition to integers.

If specified, \textit{separators} should be an \((\text{item}\_\text{separator}, \text{key}\_\text{separator})\) tuple. The default is \(('', ',', ':')\) if \( \text{indent} \) is \texttt{None} and \((', ',', ':')\) otherwise. To get the most compact JSON representation, you should specify \( ('', ',', ':') \) to eliminate whitespace.

\textit{Changed in version 3.4:} Use \((', ',', ':')\) as default if \( \text{indent} \) is not \texttt{None}.

If specified, \textit{default} should be a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a \texttt{TypeError}. If not specified, \texttt{TypeError} is raised.

If \textit{sort\_keys} is true (default: \texttt{False}), then the output of dictionaries will be sorted by key.

To use a custom \texttt{JSONEncoder} subclass (e.g. one that overrides the \texttt{default()} method to serialize additional types), specify it with the \texttt{cls} kwarg; otherwise \texttt{JSONEncoder} is used.

\textit{Changed in version 3.6:} All optional parameters are now \texttt{keyword-only}.

\textbf{Note:} Unlike \texttt{pickle} and \texttt{marshal}, JSON is not a framed
protocol, so trying to serialize multiple objects with repeated calls to `dump()` using the same `fp` will result in an invalid JSON file.

```python
json.dumps(obj, *, skipkeys=False, ensure_ascii=True,
check_circular=True, allow_nan=True, indent=None,
separators=None, default=None, sort_keys=False, **kw)
```

Serialize `obj` to a JSON formatted `str` using this conversion table. The arguments have the same meaning as in `dump()`.

**Note:** Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, `loads(dumps(x)) != x` if `x` has non-string keys.

```python
json.load(fp, *, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None,
**kw)
```

Deserialize `fp` (a `.read()`-supporting text file or binary file containing a JSON document) to a Python object using this conversion table.

`object_hook` is an optional function that will be called with the result of any object literal decoded (a `dict`). The return value of `object_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

`object_pairs_hook` is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. 
This feature can be used to implement custom decoders. If `object_hook` is also defined, the `object_pairs_hook` takes priority.

*Changed in version 3.1:* Added support for `object_pairs_hook`.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

`parse_constant`, if specified, will be called with one of the following strings: `-Infinity`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

*Changed in version 3.1:* `parse_constant` doesn’t get called on ‘null’, ‘true’, ‘false’ anymore.

To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg; otherwise `JSONDecoder` is used. Additional keyword arguments will be passed to the constructor of the class.

If the data being deserialized is not a valid JSON document, a `JSONDecodeError` will be raised.

*Changed in version 3.6:* All optional parameters are now keyword-only.

*Changed in version 3.6:* `fp` can now be a binary file. The input encoding should be UTF-8, UTF-16 or UTF-32.
json. `loads(s, *, encoding=None, cls=None, object_hook=None,
parse_float=None, parse_int=None, parse_constant=None,
object_pairs_hook=None, **kw)`

Deserializes `s` (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table.

The other arguments have the same meaning as in `load()`, except `encoding` which is ignored and deprecated.

If the data being deserialized is not a valid JSON document, a `JSONDecodeError` will be raised.

*Changed in version 3.6:* `s` can now be of type `bytes` or `bytearray`. The input encoding should be UTF-8, UTF-16 or UTF-32.
Encoders and Decoders

class json.JSONDecoder(*, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)

Simple JSON decoder.

Performs the following translations in decoding by default:

<table>
<thead>
<tr>
<th>JSON</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>dict</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>number (int)</td>
<td>int</td>
</tr>
<tr>
<td>number (real)</td>
<td>float</td>
</tr>
<tr>
<td>true</td>
<td>True</td>
</tr>
<tr>
<td>false</td>
<td>False</td>
</tr>
<tr>
<td>null</td>
<td>None</td>
</tr>
</tbody>
</table>

It also understands NaN, Infinity, and -Infinity as their corresponding float values, which is outside the JSON spec.

*object_hook*, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given dict. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

*object_pairs_hook*, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value
of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders. If `object_hook` is also defined, the `object_pairs_hook` takes priority.

*Changed in version 3.1:* Added support for `object_pairs_hook`.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

`parse_constant`, if specified, will be called with one of the following strings: `-Infinity`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

If `strict` is false (`True` is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0–31 range, including `\t` (tab), `\n`, `\r` and `\0`.

If the data being deserialized is not a valid JSON document, a `JSONDecodeError` will be raised.

*Changed in version 3.6:* All parameters are now keyword-only.

`decode(s)`

Return the Python representation of `s` (a `str` instance containing a JSON document).
JSONDecodeError will be raised if the given JSON document is not valid.

raw_decode(s)
Decode a JSON document from s (a str beginning with a JSON document) and return a 2-tuple of the Python representation and the index in s where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json. JSONEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)
Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

<table>
<thead>
<tr>
<th>Python</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>object</td>
</tr>
<tr>
<td>list, tuple</td>
<td>array</td>
</tr>
<tr>
<td>str</td>
<td>string</td>
</tr>
<tr>
<td>int, float, int- &amp; float-derived Enums</td>
<td>number</td>
</tr>
<tr>
<td>True</td>
<td>true</td>
</tr>
<tr>
<td>False</td>
<td>false</td>
</tr>
<tr>
<td>None</td>
<td>null</td>
</tr>
</tbody>
</table>

Changed in version 3.4: Added support for int- and float-derived Enum classes.

To extend this to recognize other objects, subclass and implement
a `default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

If `skipkeys` is false (the default), then it is a `TypeError` to attempt encoding of keys that are not `str`, `int`, `float` or `None`. If `skipkeys` is true, such items are simply skipped.

If `ensure_ascii` is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `ensure_ascii` is false, these characters will be output as-is.

If `check_circular` is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an `OverflowError`). Otherwise, no such check takes place.

If `allow_nan` is true (the default), then `NaN`, `Infinity`, and `-Infinity` will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a `ValueError` to encode such floats.

If `sort_keys` is true (default: `False`), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If `indent` is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If
*indent* is a string (such as "\t"), that string is used to indent each level.

*Changed in version 3.2:* Allow strings for *indent* in addition to integers.

If specified, *separators* should be an *(item_separator, key_separator)* tuple. The default is *(',', ':')* if *indent* is None and *(',', ':')* otherwise. To get the most compact JSON representation, you should specify *(',', ':')* to eliminate whitespace.

*Changed in version 3.4:* Use *(',', ':')* as default if *indent* is not None.

If specified, *default* should be a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a *TypeError*. If not specified, *TypeError* is raised.

*Changed in version 3.6:* All parameters are now *keyword-only*.

**default(o)**

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a *TypeError*).

For example, to support arbitrary iterators, you could implement default like this:

```python
def default(self, o):
    try:
        iterable = iter(o)
```
except TypeError:
    pass
else:
    return list(iterable)

# Let the base class default method raise the exception
return json.JSONEncoder.default(self, o)

**encode(o)**

Return a JSON string representation of a Python data structure, `o`. For example:

```python
generate_json_string = json.JSONEncoder().encode({'foo': ['bar', 'baz']})
print(generate_json_string)
```

**iterencode(o)**

Encode the given object, `o`, and yield each string representation as available. For example:

```python
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```
Exceptions

```
exception json. JSONDecodeError(msg, doc, pos)
```

Subclass of `ValueError` with the following additional attributes:

- **msg**
  - The unformatted error message.

- **doc**
  - The JSON document being parsed.

- **pos**
  - The start index of `doc` where parsing failed.

- **lineno**
  - The line corresponding to `pos`.

- **colno**
  - The column corresponding to `pos`.

*New in version 3.5.*
Standard Compliance and Interoperability

The JSON format is specified by RFC 7159 and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, JSONEncoder and JSONDecoder subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module's deserializer is technically RFC-compliant under default settings.

Character Encodings

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability.

As permitted, though not required, by the RFC, this module's serializer sets ensure_ascii=True by default, thus escaping the output so that the resulting strings only contain ASCII characters.

Other than the ensure_ascii parameter, this module is defined strictly in terms of conversion between Python objects and Unicode strings, and thus does not otherwise directly address the issue of
character encodings.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module's serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module's deserializer raises a \texttt{ValueError} when an initial BOM is present.

The RFC does not explicitly forbid JSON strings which contain byte sequences that don't correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original \texttt{str}) code points for such sequences.

\section*{Infinite and NaN Number Values}

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs \texttt{Infinity}, \texttt{-Infinity}, and \texttt{NaN} as if they were valid JSON number literal values:

\begin{verbatim}
>>> # Neither of these calls raises an exception, but
>>> json.dumps(float('-inf'))
'Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
\end{verbatim}

In the serializer, the \texttt{allow_nan} parameter can be used to alter this
behavior. In the deserializer, the \textit{parse\_constant} parameter can be used to alter this behavior.

\section*{Repeated Names Within an Object}

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

\begin{verbatim}
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
\end{verbatim}

The \textit{object\_pairs\_hook} parameter can be used to alter this behavior.

\section*{Top-level Non-Object, Non-Array Values}

The old version of JSON specified by the obsolete RFC 4627 required that the top-level value of a JSON text must be either a JSON object or array (Python \texttt{dict} or \texttt{list}), and could not be a JSON null, boolean, number, or string value. RFC 7159 removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

\section*{Implementation Limitations}

Some JSON deserializer implementations may set limits on:
• the size of accepted JSON texts
• the maximum level of nesting of JSON objects and arrays
• the range and precision of JSON numbers
• the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation’s range and precision limitations. This is especially relevant when serializing Python int values of extremely large magnitude, or when serializing instances of “exotic” numerical types such as decimal.Decimal.
Command Line Interface

**Source code:** Lib/json/tool.py

The `json.tool` module provides a simple command line interface to validate and pretty-print JSON objects.

If the optional `infile` and `outfile` arguments are not specified, `sys.stdin` and `sys.stdout` will be used respectively:

```bash
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1
```

*Changed in version 3.5:* The output is now in the same order as the input. Use the `--sort-keys` option to sort the output of dictionaries alphabetically by key.

**Command line options**

**infile**

The JSON file to be validated or pretty-printed:

```bash
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    }
]```
If `infile` is not specified, read from `sys.stdin`.

**outfile**

Write the output of the `infile` to the given `outfile`. Otherwise, write it to `sys.stdout`.

**--sort-keys**

Sort the output of dictionaries alphabetically by key.

*New in version 3.5.*

**-h, --help**

Show the help message.

**Footnotes**

As noted in [the errata for RFC 7159](https://tools.ietf.org/html/rfc7159), JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.
Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name “mailcap” is derived from the phrase “mail capability”.) For example, a mailcap file might contain a line like `video/mpeg; xmpeg %s`. Then, if the user encounters an email message or Web document with the MIME type `video/mpeg`, `%s` will be replaced by a filename (usually one belonging to a temporary file) and the `xmpeg` program can be automatically started to view the file.

The mailcap format is documented in RFC 1524, “A User Agent Configuration Mechanism For Multimedia Mail Format Information,” but is not an Internet standard. However, mailcap files are supported on most Unix systems.

```python
def findmatch(caps, MIMEtype, key='view', filename='/dev/null', plist=[]):
    # Implementation details...
```

Return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to `os.system()`), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, `(None, None)` is returned.

`key` is the name of the field desired, which represents the type of activity to be performed; the default value is ‘view’, since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be ‘compose’ and ‘edit’, if
you wanted to create a new body of the given MIME type or alter the existing body data. See RFC 1524 for a complete list of these fields.

*filename* is the filename to be substituted for `%s` in the command line; the default value is `/dev/null` which is almost certainly not what you want, so usually you'll override it by specifying a filename.

*plist* can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign ('='), and the parameter's value. Mailcap entries can contain named parameters like `%{foo}`, which will be replaced by the value of the parameter named 'foo'. For example, if the command line `showpartial %{id} %{number} %{total}` was in a mailcap file, and *plist* was set to `['id=1', 'number=2', 'total=3']`, the resulting command line would be `showpartial 1 2 3`.

In a mailcap file, the “test” field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. *findmatch()* will automatically check such conditions and skip the entry if the check fails.

**mailcap.getcaps()**

Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the *findmatch()* function. An entry is stored as a list of dictionaries, but it shouldn't be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the
system. Settings in the user's mailcap file `$HOME/.mailcap` will override settings in the system mailcap files `/etc/mailcap`, `/usr/etc/mailcap`, and `/usr/local/etc/mailcap`.

An example usage:

```python
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223', ('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
mailbox — Manipulate mailboxes in various formats

Source code: Lib/mailbox.py

This module defines two classes, Mailbox and Message, for accessing and manipulating on-disk mailboxes and the messages they contain. Mailbox offers a dictionary-like mapping from keys to messages. Message extends the email.message module's Message class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also:

Module email
Represent and manipulate messages.
Mailbox objects

class mailbox.Mailbox
A mailbox, which may be inspected and modified.

The Mailbox class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from Mailbox and your code should instantiate a particular subclass.

The Mailbox interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the Mailbox instance with which they will be used and are only meaningful to that Mailbox instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a Mailbox instance using the set-like method add() and removed using a del statement or the set-like methods remove() and discard().

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a Message instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a Mailbox instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the Mailbox instance.

The default Mailbox iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and
well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

**Warning:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you’re modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

**add**(message)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it’s an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

*Changed in version 3.2:* Support for binary input was added.
remove(key)
__delitem__(key)
discard(key)

Delete the message corresponding to key from the mailbox.

If no such message exists, a KeyError exception is raised if the method was called as remove() or __delitem__() but no exception is raised if the method was called as discard(). The behavior of discard() may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(key, message)

Replace the message corresponding to key with message. Raise a KeyError exception if no message already corresponds to key.

As with add(), parameter message may be a Message instance, an email.message.Message instance, a string, a byte string, or a file-like object (which should be open in binary mode). If message is an instance of the appropriate format-specific Message subclass (e.g., if it's an mboxMessage instance and this is an mbox instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to key is left unchanged.

iterkeys()
keys()

Return an iterator over all keys if called as iterkeys() or return a list of keys if called as keys().
**itervalues()**

**__iter__()**

**values()**

Return an iterator over representations of all messages if called as `itervalues()` or `__iter__()` or return a list of such representations if called as `values()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

**Note:** The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

**iteritems()**

**items()**

Return an iterator over `(key, message)` pairs, where `key` is a key and `message` is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

**get(key, default=None)**

**__getitem__(key)**

Return a representation of the message corresponding to `key`. If no such message exists, `default` is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was
specified when the Mailbox instance was initialized.

**get_message**(key)

Return a representation of the message corresponding to key as an instance of the appropriate format-specific Message subclass, or raise a KeyError exception if no such message exists.

**get_bytes**(key)

Return a byte representation of the message corresponding to key, or raise a KeyError exception if no such message exists.

*New in version 3.2.*

**get_string**(key)

Return a string representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The message is processed through email.message.Message to convert it to a 7bit clean representation.

**get_file**(key)

Return a file-like representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

*Changed in version 3.2:* The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context management protocol: you can use a with statement to automatically close it.
Note: Unlike other representations of messages, file-like representations are not necessarily independent of the Mailbox instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(key)
Return True if key corresponds to a message, False otherwise.

__len__()  
Return a count of messages in the mailbox.

clear()  
Delete all messages from the mailbox.

pop(key, default=None)
Return a representation of the message corresponding to key and delete the message. If no such message exists, return default. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

popitem()
Return an arbitrary (key, message) pair, where key is a key and message is a message representation, and delete the corresponding message. If the mailbox is empty, raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.
**update**(arg)
Parameter arg should be a *key-to-message* mapping or an iterable of *(key, message)* pairs. Updates the mailbox so that, for each given key and message, the message corresponding to key is set to message as if by using *__setitem__()*(). As with *__setitem__()*(), each key must already correspond to a message in the mailbox or else a *KeyError* exception will be raised, so in general it is incorrect for arg to be a *Mailbox* instance.

>Note: Unlike with dictionaries, keyword arguments are not supported.

**flush()**
Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and flush() does nothing, but you should still make a habit of calling this method.

**lock()**
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

**unlock()**
Release the lock on the mailbox, if any.

**close()**
Flush the mailbox, unlock it if necessary, and close any open
files. For some Mailbox subclasses, this method does nothing.

**Maildir**

class mailbox.Maildir(dirname, factory=None, create=True)

A subclass of Mailbox for mailboxes in Maildir format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MaildirMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

It is for historical reasons that dirname is named as such rather than path.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special “info” section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is
considered a folder if "." is the first character in its name. Folder names are represented by Maildir without the leading ".". Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using "." to delimit levels, e.g., “Archived.2005.07”.

Note: The Maildir specification requires the use of a colon (':') in certain message file names. However, some operating systems do not permit this character in file names. If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (':'') is a popular choice. For example:

```python
import mailbox
mailbox.Maildir.colon = '!
```

The colon attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of Mailbox in addition to the following:

**list_folders()**

Return a list of the names of all folders.

**get_folder(folder)**

Return a Maildir instance representing the folder whose name is `folder`. A NoSuchMailboxError exception is raised if the folder does not exist.

**add_folder(folder)**

Create a folder whose name is `folder` and return a Maildir instance representing it.
remove_folder(folder)
Delete the folder whose name is folder. If the folder contains any messages, aNotEmptyError exception will be raised and the folder will not be deleted.

clean()
Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some Mailbox methods implemented by Maildir deserve special remarks:

add(message)
__setitem__(key, message)
update(arg)

**Warning:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
Maildir mailboxes do not support (or require) locking, so these methods do nothing.
**close()**

Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

**get_file(key)**

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

**See also:**

*maildir man page from qmail*  
The original specification of the format.

**Using maildir format**

Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on “info” semantics.

*maildir man page from Courier*  
Another specification of the format. Describes a common extension for supporting folders.

**mbox**

`class mailbox. mbox(path, factory=None, create=True)`  
A subclass of Mailbox for mailboxes in mbox format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, `mboxMessage` is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.
The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are “From “.

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, mbox implements the original format, which is sometimes referred to as mboxo. This means that the Content-Length header, if present, is ignored and that any occurrences of “From ” at the beginning of a line in a message body are transformed to “>From ” when storing the message, although occurrences of “>From ” are not transformed to “From ” when reading the message.

Some Mailbox methods implemented by mbox deserve special remarks:

get_file(key)
   Using the file after calling flush() or close() on the mbox instance may yield unpredictable results or raise an exception.

lock()
unlock()
   Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See also:

mbox man page from qmail
   A specification of the format and its variations.

mbox man page from tin
   Another specification of the format, with details on locking.
Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad

An argument for using the original mbox format rather than a variation.

“mbox” is a family of several mutually incompatible mailbox formats

A history of mbox variations.

MH

class mailbox. **MH**(path, factory=None, create=True)

A subclass of Mailbox for mailboxes in MH format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MHMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called folders) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support sequences, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of mh’s behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by mh to store its state and configuration.
MH instances have all of the methods of Mailbox in addition to the following:

**list_folders()**
Return a list of the names of all folders.

**get_folder(folder)**
Return an MH instance representing the folder whose name is *folder*. A NoSuchMailboxError exception is raised if the folder does not exist.

**add_folder(folder)**
Create a folder whose name is *folder* and return an MH instance representing it.

**remove_folder(folder)**
Delete the folder whose name is *folder*. If the folder contains any messages, a NotEmptyError exception will be raised and the folder will not be deleted.

**get_sequences()**
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

**set_sequences(sequences)**
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by get_sequences().

**pack()**
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated.
correspondingly.

**Note:** Already-issued keys are invalidated by this operation and should not be subsequently used.

Some **Mailbox** methods implemented by **MH** deserve special remarks:

```python
remove(key)
__delitem__(key)
discard(key)
```

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

```python
lock()
unlock()
```

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

```python
get_file(key)
```

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

```python
flush()
```

All changes to MH mailboxes are immediately applied, so this method does nothing.

```python
close()
```
MH instances do not keep any open files, so this method is equivalent to `unlock()`.

**See also:**

**nmh - Message Handling System**
Home page of nmh, an updated version of the original mh.

**MH & nmh: Email for Users & Programmers**
A GPL-licensed book on mh and nmh, with some information on the mailbox format.

**Babyl**

```python
class mailbox. Babyl(path, factory=None, create=True)
```

A subclass of `Mailbox` for mailboxes in Babyl format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is `None`, `BabylMessage` is used as the default message representation. If `create` is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically
a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of Mailbox in addition to the following:

**get_labels()**
Return a list of the names of all user-defined labels used in the mailbox.

**Note:** The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some Mailbox methods implemented by Babyl deserve special remarks:

**get_file**(key)
In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an `io.BytesIO` instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

**lock()**
**unlock()**
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also:

**Format of Version 5 Babyl Files**
A specification of the Babyl format.

**Reading Mail with Rmail**
The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.MMDF(path, factory=None, create=True)
A subclass of Mailbox for mailboxes in MMDF format. Parameter `factory` is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If `factory` is None, MMDFMessage is used as the default message representation. If `create` is True, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`\001`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are “From “, but additional occurrences of “From ” are not transformed to “>From ” when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.
Some **Mailbox** methods implemented by **MMDF** deserve special remarks:

**get_file**(key)

Using the file after calling flush() or close() on the **MMDF** instance may yield unpredictable results or raise an exception.

**lock()**

**unlock()**

Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

**See also:**

**mmdf man page from tin**

A specification of MMDF format from the documentation of tin, a newsreader.

**MMDF**

A Wikipedia article describing the Multichannel Memorandum Distribution Facility.
Message objects

class mailbox.Message(message=None)


If message is omitted, the new instance is created in a default, empty state. If message is an email.message.Message instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if message is a Message instance. If message is a string, a byte string, or a file, it should contain an RFC 2822-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that Message instances be used to represent messages retrieved using Mailbox instances. In some situations, the time and memory required to generate Message representations might not be acceptable. For such situations, Mailbox instances also offer string and file-like representations,
and a custom message factory may be specified when a Mailbox instance is initialized.

**MaildirMessage**

class mailbox.MaildirMessage(message=None)

A message with Maildir-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in cur has an “info” section added to its file name to store information about its state. (Some mail readers may also add an “info” section to messages in new.) The “info” section may take one of two forms: it may contain “2,” followed by a list of standardized flags (e.g., “2,FR”) or it may contain “1,” followed by so-called experimental information. Standard flags for Maildir messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Draft</td>
<td>Under composition</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>P</td>
<td>Passed</td>
<td>Forwarded, resent, or bounced</td>
</tr>
<tr>
<td>R</td>
<td>Replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>S</td>
<td>Seen</td>
<td>Read</td>
</tr>
<tr>
<td>T</td>
<td>Trashed</td>
<td>Marked for subsequent deletion</td>
</tr>
</tbody>
</table>

MaildirMessage instances offer the following methods:
get_subdir()
Return either “new” (if the message should be stored in the new subdirectory) or “cur” (if the message should be stored in the cur subdirectory).

Note: A message is typically moved from new to cur after its mailbox has been accessed, whether or not the message is has been read. A message msg has been read if "S" in msg.get_flags() is True.

set_subdir(subdir)
Set the subdirectory the message should be stored in. Parameter subdir must be either “new” or “cur”.

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of 'D', 'F', 'P', 'R', 'S', and 'T'. The empty string is returned if no flags are set or if “info” contains experimental semantics.

set_flags(flags)
Set the flags specified by flags and unset all others.

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character. The current “info” is overwritten whether or not it contains experimental information rather than flags.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character. If “info” contains experimental information rather than flags, the current “info” is not modified.

**get_date()**
Return the delivery date of the message as a floating-point number representing seconds since the epoch.

**set_date(date)**
Set the delivery date of the message to date, a floating-point number representing seconds since the epoch.

**get_info()**
Return a string containing the “info” for a message. This is useful for accessing and modifying “info” that is experimental (i.e., not a list of flags).

**set_info(info)**
Set “info” to info, which should be a string.

When a MaildirMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>O flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>R flag</td>
<td>A flag</td>
</tr>
<tr>
<td>S flag</td>
<td>R flag</td>
</tr>
<tr>
<td>T flag</td>
<td>D flag</td>
</tr>
</tbody>
</table>
When a `MaildirMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>“cur” subdirectory and S flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>R flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When a `MaildirMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cur” subdirectory</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>“cur” subdirectory and S flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>P flag</td>
<td>“forwarded” or “resent” label</td>
</tr>
<tr>
<td>R flag</td>
<td>“answered” label</td>
</tr>
<tr>
<td>T flag</td>
<td>“deleted” label</td>
</tr>
</tbody>
</table>

**mboxMessage**

class mailbox.mboxMessage(message=None)

A message with mbox-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable
variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in Status and X-Status headers.

Conventional flags for mbox messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The “R” and “O” flags are stored in the Status header, and the “D”, “F”, and “A” flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

**get_from()**

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

**set_from(from_, time_=None)**

Set the “From ” line to from_, which should be specified without a leading “From ” or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to
time.strftime(), or \texttt{True} (to use \texttt{time.gmtime()}).

\textbf{get\_flags()}

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

\textbf{set\_flags(flags)}

Set the flags specified by \texttt{flags} and unset all others. Parameter \texttt{flags} should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

\textbf{add\_flag(flag)}

Set the flag(s) specified by \texttt{flag} without changing other flags. To add more than one flag at a time, \texttt{flag} may be a string of more than one character.

\textbf{remove\_flag(flag)}

Unset the flag(s) specified by \texttt{flag} without changing other flags. To remove more than one flag at a time, \texttt{flag} maybe a string of more than one character.

When an \texttt{mboxMessage} instance is created based upon a \texttt{MaildirMessage} instance, a “From ” line is generated based upon the \texttt{MaildirMessage} instance's delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
</tbody>
</table>
When an `mboxMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When an `mboxMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>D flag</td>
<td>“deleted” label</td>
</tr>
<tr>
<td>A flag</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

When a `Message` instance is created based upon an `MMDFMessage` instance, the “From ” line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>

**MHMessage**

*class* mailbox. **MHMessage**(message=None)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard *mh* and *nmh*) use sequences in much the same way flags are used with other formats, as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>flagged</td>
<td>Marked as important</td>
</tr>
</tbody>
</table>

**MHMessage** instances offer the following methods:

**get_sequences()**

Return a list of the names of sequences that include this message.

**set_sequences(sequences)**

Set the list of sequences that include this message.

**add_sequence(sequence)**
Add *sequence* to the list of sequences that include this message.

**remove_sequence**(sequence)

Remove *sequence* from the list of sequences that include this message.

When an *MHMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no S flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>R flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an *MHMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no R flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>A flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an *MHMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“unseen” sequence | “unseen” label
---|---
“replied” sequence | “answered” label

**BabylMessage**

`class mailbox.BabylMessage(message=None)`

A message with Babyl-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>filed</td>
<td>Copied to another file or mailbox</td>
</tr>
<tr>
<td>answered</td>
<td>Replied to</td>
</tr>
<tr>
<td>forwarded</td>
<td>Forwarded</td>
</tr>
<tr>
<td>edited</td>
<td>Modified by the user</td>
</tr>
<tr>
<td>resent</td>
<td>Resent</td>
</tr>
</tbody>
</table>

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

**get_labels()**

Return a list of labels on the message.
**set_labels**(*labels*)

Set the list of labels on the message to *labels*.

**add_label**(*label*)

Add *label* to the list of labels on the message.

**remove_label**(*label*)

Remove *label* from the list of labels on the message.

**get_visible**()

Return an *Message* instance whose headers are the message's visible headers and whose body is empty.

**set_visible**(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

**update_visible**()

When a *BabylMessage* instance’s original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date, From, Reply-To, To, CC, and Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:
<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no S flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>T flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>R flag</td>
</tr>
<tr>
<td>“forwarded” label</td>
<td>P flag</td>
</tr>
</tbody>
</table>

When a `BabylMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no R flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>D flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>A flag</td>
</tr>
</tbody>
</table>

When a `BabylMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>“answered” label</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

**MMDFMessage**

class mailbox.MMDFMessage(message=None)

A message with MMDF-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

As with message in an `mbox` mailbox, MMDF messages are stored...
with the sender’s address and the delivery date in an initial line beginning with “From “. Likewise, flags that indicate the state of the message are typically stored in \textit{Status} and \textit{X-Status} headers.

Conventional flags for MMDF messages are identical to those of \textit{mbox} message and are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The “R” and “O” flags are stored in the \textit{Status} header, and the “D”, “F”, and “A” flags are stored in the \textit{X-Status} header. The flags and headers typically appear in the order mentioned.

\texttt{MMDFMessage} instances offer the following methods, which are identical to those offered by \texttt{mboxMessage}:

\texttt{get\_from()}

Return a string representing the “From ” line that marks the start of the message in an \texttt{mbox} mailbox. The leading “From ” and the trailing newline are excluded.

\texttt{set\_from(from\_, time\_=None)}

Set the “From ” line to \texttt{from\_}, which should be specified without a leading “From ” or trailing newline. For convenience, \texttt{time\_} may be specified and will be formatted appropriately and appended to \texttt{from\_}. If \texttt{time\_} is specified, it should be a
time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).

goingflags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of 'R', 'O', 'D', 'F', and 'A'.

set_flags(flags)
Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)
Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character.

remove_flag(flag)
Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character.

When an MMDFMessage instance is created based upon a MaildirMessage instance, a “From ” line is generated based upon the MaildirMessage instance's delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>D flag</td>
<td>“deleted” label</td>
</tr>
<tr>
<td>A flag</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the “From ” line is copied and all flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>
Exceptions

The following exception classes are defined in the `mailbox` module:

`exception mailbox.Error`
The based class for all other module-specific exceptions.

`exception mailbox.NoSuchMailboxError`
Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the `create` parameter set to `False`), or when opening a folder that does not exist.

`exception mailbox.NotEmptyError`
Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

`exception mailbox.ExternalClashError`
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

`exception mailbox.FormatError`
Raised when the data in a file cannot be parsed, such as when an `MH` instance attempts to read a corrupted `.mh_sequences` file.
Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```python
import mailbox
for message in mailbox.mbox('~/mbox '):
    subject = message['subject']  # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```python
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```python
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bug')
```
```python
boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue  # The message is malformed.
    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break  # Found destination,

for box in boxes.itervalues():
    box.close()
```
mimetypes — Map filenames to MIME types

Source code: Lib/mimetypes.py

The mimetypes module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call init() if they rely on the information init() sets up.

mimetypes.guess_type(url, strict=True)

Guess the type of a file based on its filename or URL, given by url. The return value is a tuple (type, encoding) where type is None if the type can’t be guessed (missing or unknown suffix) or a string of the form 'type/subtype', usable for a MIME content-type header.

encoding is None for no encoding or the name of the program used to encode (e.g. compress or gzip). The encoding is suitable for use as a Content-Encoding header, not as a Content-Transfer-Encoding header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case
sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types *registered with IANA*. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

**mimetypes.** `guess_all_extensions(type, strict=True)`

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`.'). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

**mimetypes.** `guess_extension(type, strict=True)`

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`.'). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()`. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

Some additional functions and data items are available for controlling the behavior of the module.
mimetypes.\texttt{\texttt{init}(files=None)}

Initialize the internal data structures. If given, \texttt{files} must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from \texttt{knownfiles}; on Windows, the current registry settings are loaded. Each file named in \texttt{files} or \texttt{knownfiles} takes precedence over those named before it. Calling \texttt{init()} repeatedly is allowed.

Specifying an empty list for \texttt{files} will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

\textit{Changed in version 3.2:} Previously, Windows registry settings were ignored.

mimetypes.\texttt{\texttt{read_mime_types}(filename)}

Load the type map given in the file \texttt{filename}, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (\texttt{'.'}), to strings of the form \texttt{'type/subtype'}. If the file \texttt{filename} does not exist or cannot be read, \texttt{None} is returned.

mimetypes.\texttt{\texttt{add_type}(type, ext, strict=True)}

Add a mapping from the MIME type \texttt{type} to the extension \texttt{ext}. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When \texttt{strict} is \texttt{True} (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.\texttt{\texttt{inited}}
Flag indicating whether or not the global data structures have been initialized. This is set to True by init().

mimetypes.**knownfiles**
List of type map file names commonly installed. These files are typically named mime.types and are installed in different locations by different packages.

mimetypes.**suffix_map**
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the .tgz extension is mapped to .tar.gz to allow the encoding and type to be recognized separately.

mimetypes.**encodings_map**
Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**
Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```python
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
```
'gzip'

>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
MimeTypes Objects

The **MimeTypes** class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the **mimetypes** module.

```python
class mimetypes.MimeTypes(filenames=(), strict=True)
```

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional `filenames` parameter can be used to cause additional files to be loaded “on top” of the default database.

**suffix_map**

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global `suffix_map` defined in the module.

**encodings_map**

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global `encodings_map` defined in the module.
**types_map**

Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

**types_map_inv**

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

**guess_extension**(type, strict=True)

Similar to the `guess_extension()` function, using the tables stored as part of the object.

**guess_type**(url, strict=True)

Similar to the `guess_type()` function, using the tables stored as part of the object.

**guess_all_extensions**(type, strict=True)

Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.

**read**(filename, strict=True)

Load MIME information from a file named `filename`. This uses `readfp()` to parse the file.

If `strict` is `True`, information will be added to list of standard types, else to the list of non-standard types.

**readfp**(fp, strict=True)
Load MIME type information from an open file `fp`. The file must have the format of the standard `mime.types` files.

If `strict` is `True`, information will be added to the list of standard types, else to the list of non-standard types.

`read_windows_registry(strict=True)`
Load MIME type information from the Windows registry.

**Availability:** Windows.

If `strict` is `True`, information will be added to the list of standard types, else to the list of non-standard types.

*New in version 3.2.*
base64 — Base16, Base32, Base64, Base85 Data Encodings

Source code: Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. It provides encoding and decoding functions for the encodings specified in RFC 3548, which defines the Base16, Base32, and Base64 algorithms, and for the de-facto standard Ascii85 and Base85 encodings.

The RFC 3548 encodings are suitable for encoding binary data so that it can safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the uuencode program.

There are two interfaces provided by this module. The modern interface supports encoding bytes-like objects to ASCII bytes, and decoding bytes-like objects or strings containing ASCII to bytes. Both base-64 alphabets defined in RFC 3548 (normal, and URL- and filesystem-safe) are supported.

The legacy interface does not support decoding from strings, but it does provide functions for encoding and decoding to and from file objects. It only supports the Base64 standard alphabet, and it adds newlines every 76 characters as per RFC 2045. Note that if you are looking for RFC 2045 support you probably want to be looking at the email package instead.
Changed in version 3.3: ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface.

Changed in version 3.4: Any bytes-like objects are now accepted by all encoding and decoding functions in this module. Ascii85/Base85 support added.

The modern interface provides:

\[
\text{base64.} \textbf{b64encode}(s, altchars=None) \\
\text{Encode the bytes-like object } s \text{ using Base64 and return the encoded bytes.}
\]

Optional altchars must be a bytes-like object of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is None, for which the standard Base64 alphabet is used.

\[
\text{base64.} \textbf{b64decode}(s, altchars=None, validate=False) \\
\text{Decode the Base64 encoded bytes-like object or ASCII string } s \text{ and return the decoded bytes.}
\]

Optional altchars must be a bytes-like object or ASCII string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

A \texttt{binascii.Error} exception is raised if \( s \) is incorrectly padded.

If \texttt{validate} is False (the default), characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check. If \texttt{validate} is True, these non-alphabet characters in the input result in a \texttt{binascii.Error}. 
base64. **standard_b64encode**(*s*)

Encode **bytes-like object** *s* using the standard Base64 alphabet and return the encoded **bytes**.

base64. **standard_b64decode**(*s*)

Decode **bytes-like object** or ASCII string *s* using the standard Base64 alphabet and return the decoded **bytes**.

base64. **urlsafe_b64encode**(*s*)

Encode **bytes-like object** *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the encoded **bytes**. The result can still contain `=`.

base64. **urlsafe_b64decode**(*s*)

Decode **bytes-like object** or ASCII string *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the decoded **bytes**.

base64. **b32encode**(*s*)

Encode the **bytes-like object** *s* using Base32 and return the encoded **bytes**.

base64. **b32decode**(*s*, **casefold**=False, **map01**=None)

Decode the Base32 encoded **bytes-like object** or ASCII string *s* and return the decoded **bytes**.

Optional **casefold** is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

**RFC 3548** allows for optional mapping of the digit 0 (zero) to the
letter O (oh), and for optional mapping of the digit 1 (one) to either
the letter l (eye) or letter L (el). The optional argument map01 when
not None, specifies which letter the digit 1 should be mapped to
(when map01 is not None, the digit 0 is always mapped to the
letter O). For security purposes the default is None, so that 0 and 1
are not allowed in the input.

A binascii.Error is raised if s is incorrectly padded or if there
are non-alphabet characters present in the input.

base64. b16encode(s)
Encode the bytes-like object s using Base16 and return the
encoded bytes.

base64. b16decode(s, casefold=False)
Decode the Base16 encoded bytes-like object or ASCII string s
and return the decoded bytes.

Optional casefold is a flag specifying whether a lowercase alphabet
is acceptable as input. For security purposes, the default is False.

A binascii.Error is raised if s is incorrectly padded or if there
are non-alphabet characters present in the input.

base64. a85encode(b, *, foldspaces=False, wrapcol=0,
pad=False, adobe=False)
Encode the bytes-like object b using Ascii85 and return the
encoded bytes.

foldspaces is an optional flag that uses the special short sequence
‘y’ instead of 4 consecutive spaces (ASCII 0x20) as supported by
‘btoa’. This feature is not supported by the “standard” Ascii85
encoding.
wrapcol controls whether the output should have newline (b'\n') characters added to it. If this is non-zero, each output line will be at most this many characters long.

pad controls whether the input is padded to a multiple of 4 before encoding. Note that the b\toa implementation always pads.

adobe controls whether the encoded byte sequence is framed with <~ and ~> , which is used by the Adobe implementation.

New in version 3.4.

base64 . a85decode(b, *, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')
Decode the Ascii85 encoded bytes-like object or ASCII string b and return the decoded bytes.

foldspaces is a flag that specifies whether the ‘y’ short sequence should be accepted as shorthand for 4 consecutive spaces (ASCII 0x20). This feature is not supported by the “standard” Ascii85 encoding.

adobe controls whether the input sequence is in Adobe Ascii85 format (i.e. is framed with <~ and ~>).

ignorechars should be a bytes-like object or ASCII string containing characters to ignore from the input. This should only contain whitespace characters, and by default contains all whitespace characters in ASCII.

New in version 3.4.

base64 . b85encode(b, pad=False)
Encode the bytes-like object b using base85 (as used in e.g.
style binary diffs) and return the encoded bytes.

If pad is true, the input is padded with b'\0' so its length is a multiple of 4 bytes before encoding.

*New in version 3.4.*

**base64.** `b85decode(b)`  
Decode the base85-encoded bytes-like object or ASCII string b and return the decoded bytes. Padding is implicitly removed, if necessary.

*New in version 3.4.*

The legacy interface:

**base64.** `decode(input, output)`  
Decode the contents of the binary input file and write the resulting binary data to the output file. input and output must be file objects. input will be read until input.readline() returns an empty bytes object.

**base64.** `decodebytes(s)`  
Decode the bytes-like object s, which must contain one or more lines of base64 encoded data, and return the decoded bytes.

*New in version 3.1.*

**base64.** `decodestring(s)`  
Deprecated alias of decodebytes().

*Deprecated since version 3.1.*

**base64.** `encode(input, output)`
Encode the contents of the binary input file and write the resulting base64 encoded data to the output file. input and output must be file objects. input will be read until input.read() returns an empty bytes object. encode() inserts a newline character (b'\n') after every 76 bytes of the output, as well as ensuring that the output always ends with a newline, as per RFC 2045 (MIME).

base64. **encodebytes**(s)

Encode the bytes-like object s, which can contain arbitrary binary data, and return bytes containing the base64-encoded data, with newlines (b'\n') inserted after every 76 bytes of output, and ensuring that there is a trailing newline, as per RFC 2045 (MIME).

*New in version 3.1.*

base64. **encodestring**(s)

Deprecated alias of encodebytes().

*Deprecated since version 3.1.*

An example usage of the module:

```python
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

**See also:**

Module binascii
Support module containing ASCII-to-binary and binary-to-ASCII conversions.

**RFC 1521** - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies

Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.
binhex — Encode and decode binhex4 files

Source code: Lib/binhex.py

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. Only the data fork is handled.

The binhex module defines the following functions:

binhex. **binhex**(input, output)

Convert a binary file with filename input to binhex file output. The output parameter can either be a filename or a file-like object (any object supporting a write() and close() method).

binhex. **hexbin**(input, output)

Decode a binhex file input. input may be a filename or a file-like object supporting read() and close() methods. The resulting file is written to a file named output, unless the argument is None in which case the output filename is read from the binhex file.

The following exception is also defined:

**exception** binhex. **Error**

Exception raised when something can't be encoded using the binhex format (for example, a filename is too long to fit in the filename field), or when input is not properly encoded binhex data.

See also:
Module binascii

Support module containing ASCII-to-binary and binary-to-ASCII conversions.
Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the old Macintosh newline convention (carriage-return as end of line).
Binascii — Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like uu, base64, or binhex instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

**Note:** a2b_* functions accept Unicode strings containing only ASCII characters. Other functions only accept bytes-like objects (such as bytes, bytearray and other objects that support the buffer protocol).

*Changed in version 3.3:* ASCII-only unicode strings are now accepted by the a2b_* functions.

The binascii module defines the following functions:

**binascii.a2b_uu(string)**

Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

**binascii.b2a_uu(data, *, backtick=False)**

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of data should be at most 45. If backtick is true, zeros are represented by
instead of spaces.

*Changed in version 3.7:* Added the *backtick* parameter.

binascii.**a2b_base64**(string)

Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

binascii.**b2a_base64**(data, *, newline=True)

Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char if *newline* is true. The output of this function conforms to *RFC 3548*.

*Changed in version 3.6:* Added the *newline* parameter.

binascii.**a2b_qp**(data, header=False)

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.

binascii.**b2a_qp**(data, quotetabs=False, istext=True, header=False)

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per *RFC 1522*. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.
**binascii.**a2b_hqx**(string)**

Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

**binascii.**rledecode_hqx**(data)**

Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses \(0x90\) after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of \(0x90\). The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the **Incomplete** exception is raised.

*Changed in version 3.2:* Accept only bytestring or bytearray objects as input.

**binascii.**rlecode_hqx**(data)**

Perform binhex4 style RLE-compression on data and return the result.

**binascii.**b2a_hqx**(data)**

Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

**binascii.**crc_hqx**(data, value)**

Compute a 16-bit CRC value of data, starting with value as the initial CRC, and return the result. This uses the CRC-CCITT polynomial \(x^{16} + x^{12} + x^{5} + 1\), often represented as 0x1021. This CRC is used in the binhex4 format.
Compute CRC-32, the 32-bit checksum of \( data \), starting with an initial CRC of \( value \). The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```python
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

*Changed in version 3.0:* The result is always unsigned. To generate the same numeric value across all Python versions and platforms, use `crc32(data) & 0xffffffff`.

**binascii.crc32**
- **data**: The binary data to compute the checksum for.
- **value**: The initial CRC value (default is 0).

**binascii.b2a_hex**
- **data**: The binary data to convert to hexadecimal.

**binascii.hexlify**
- **data**: The binary data to convert to hexadecimal.

**binascii.a2b_hex**
- **hexstr**: The hexadecimal string to convert to binary.

**binascii.unhexlify**
- **hexstr**: The hexadecimal string to convert to binary.
Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the `bytes.fromhex()` class method.

`exception` `binascii.Error`

Exception raised on errors. These are usually programming errors.

`exception` `binascii.Incomplete`

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

**See also:**

**Module** `base64`

Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

**Module** `binhex`

Support for the binhex format used on the Macintosh.

**Module** `uu`

Support for UU encoding used on Unix.

**Module** `quopri`

Support for quoted-printable encoding used in MIME email messages.
quopri — Encode and decode MIME quoted-printable data

Source code: Lib/quopri.py

This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the base64 module is more compact if there are many such characters, as when sending a graphics file.

quopri.decode(input, output, header=False)

Decode the contents of the input file and write the resulting decoded binary data to the output file. input and output must be binary file objects. If the optional argument header is present and true, underscore will be decoded as space. This is used to decode “Q”-encoded headers as described in RFC 1522: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text”.

quopri.encode(input, output, quotetabs, header=False)

Encode the contents of the input file and write the resulting quoted-printable data to the output file. input and output must be binary file objects. quotetabs, a non-optional flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded.
Note that spaces and tabs appearing at the end of lines are always encoded, as per RFC 1521. header is a flag which controls if spaces are encoded as underscores as per RFC 1522.

quopri. **decodestring**(s, header=False)
   Like decode(), except that it accepts a source bytes and returns the corresponding decoded bytes.

quopri. **encodestring**(s, quotetabs=False, header=False)
   Like encode(), except that it accepts a source bytes and returns the corresponding encoded bytes. By default, it sends a False value to quotetabs parameter of the encode() function.

See also:

Module base64
   Encode and decode MIME base64 data
uu — Encode and decode uuencode files

Source code: Lib/uu.py

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname ' - ' is understood to mean the standard input or output. However, this interface is deprecated; it's better for the caller to open the file itself, and be sure that, when required, the mode is 'rb' or 'wb' on Windows.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The uu module defines the following functions:

uu.encode(in_file, out_file, name=None, mode=None, *, backtick=False)

Uuencode file in_file into file out_file. The uuencoded file will have the header specifying name and mode as the defaults for the results of decoding the file. The default defaults are taken from in_file, or ' - ' and 0o666 respectively. If backtick is true, zeros are represented by ' ` ' instead of spaces.

Changed in version 3.7: Added the backtick parameter.
```
uu.decode(in_file, out_file=None, mode=None, quiet=False)
```

This call decodes uuencoded file `in_file` placing the result on file `out_file`. If `out_file` is a pathname, `mode` is used to set the permission bits if the file must be created. Defaults for `out_file` and `mode` are taken from the uuencode header. However, if the file specified in the header already exists, a `uu.Error` is raised.

`decode()` may print a warning to standard error if the input was produced by an incorrect uuencoder and Python could recover from that error. Setting `quiet` to a true value silences this warning.

**exception uu.Error**

Subclass of `Exception`, this can be raised by `uu.decode()` under various situations, such as described above, but also including a badly formatted header, or truncated input file.

**See also:**

**Module binascii**

Support module containing ASCII-to-binary and binary-to-ASCII conversions.
Structured Markup Processing Tools

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

- **html** — HyperText Markup Language support
- **html.parser** — Simple HTML and XHTML parser
  - Example HTML Parser Application
  - HTMLParser Methods
  - Examples
- **html.entities** — Definitions of HTML general entities
- **XML Processing Modules**
  - XML vulnerabilities
  - The defusedxml and defusedexpat Packages
- **xml.etree.ElementTree** — The ElementTree XML API
  - Tutorial
  - XML tree and elements
  - Parsing XML
  - Pull API for non-blocking parsing
  - Finding interesting elements
  - Modifying an XML File
  - Building XML documents
  - Parsing XML with Namespaces
  - Additional resources
  - XPath support
    - Example
    - Supported XPath syntax
- Reference
  - Functions
  - Element Objects
  - ElementTree Objects
  - QName Objects
  - TreeBuilder Objects
  - XMLParser Objects
  - XMLPullParser Objects
  - Exceptions
- `xml.dom` — The Document Object Model API
  - Module Contents
  - Objects in the DOM
    - DOMImplementation Objects
    - Node Objects
    - NodeList Objects
    - DocumentType Objects
    - Document Objects
    - Element Objects
    - Attr Objects
    - NamedNodeMap Objects
    - Comment Objects
    - Text and CDATASection Objects
    - ProcessingInstruction Objects
    - Exceptions
  - Conformance
    - Type Mapping
    - Accessor Methods
- `xml.dom.minidom` — Minimal DOM implementation
  - DOM Objects
  - DOM Example
  - minidom and the DOM standard
- `xml.dom.pulldom` — Support for building partial DOM trees
  - DOMEventStream Objects
- `xml.sax` — Support for SAX2 parsers
- SAXException Objects
- `xml.sax.handler` — Base classes for SAX handlers
  - ContentHandler Objects
  - DTDHandler Objects
  - EntityResolver Objects
  - ErrorHandler Objects
- `xml.sax.saxutils` — SAX Utilities
- `xml.sax.xmlreader` — Interface for XML parsers
  - XMLReader Objects
  - IncrementalParser Objects
  - Locator Objects
  - InputSource Objects
  - The Attributes Interface
  - The AttributesNS Interface
- `xml.parsers.expat` — Fast XML parsing using Expat
  - XMLParser Objects
  - ExpatError Exceptions
  - Example
  - Content Model Descriptions
  - Expat error constants
This module defines utilities to manipulate HTML.

```python
html.escape(s, quote=True)
```

Convert the characters `&`, `<` and `>` in string `s` to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag `quote` is true, the characters `"` and `'` are also translated; this helps for inclusion in an HTML attribute value delimited by quotes, as in `<a href="...">`.

*New in version 3.2.*

```python
html.unescape(s)
```

Convert all named and numeric character references (e.g. `&gt;`, `&#62;`, `&#x3e;`) in the string `s` to the corresponding Unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the list of HTML 5 named character references.

*New in version 3.4.*

Submodules in the `html` package are:

- `html.parser` – HTML/XHTML parser with lenient parsing mode
**html.parser** — Simple HTML and XHTML parser

**Source code:** Lib/html/parser.py

This module defines a class `HTMLParser` which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

```python
class html.parser.HTMLParser(*, convert_charrefs=True)
```

Create a parser instance able to parse invalid markup.

If `convert_charrefs` is `True` (the default), all character references (except the ones in `script/style` elements) are automatically converted to the corresponding Unicode characters.

An `HTMLParser` instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass `HTMLParser` and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

*Changed in version 3.4:* `convert_charrefs` keyword argument added.

*Changed in version 3.5:* The default value for argument
convert_charrefs is now True.
Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the HTMLParser class to print out start tags, end tags, and data as they are encountered:

```python
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag:", tag)

    def handle_data(self, data):
        print("Encountered some data:", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>
<body><h1>Parse me!</h1></body></html>

The output will then be:

Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data: Test
Encountered an end tag: title
Encountered an end tag: head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data: Parse me!
Encountered an end tag: h1
Encountered an end tag: body
Encountered an end tag: html
**HTMLParser Methods**

**HTMLParser** instances have the following methods:

**HTMLParser**. `feed(data)`
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called. `data` must be `str`.

**HTMLParser**. `close()`
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the **HTMLParser** base class method `close()`.

**HTMLParser**. `reset()`
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

**HTMLParser**. `getpos()`
Return current line number and offset.

**HTMLParser**. `get_starttag_text()`
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The
base class implementations do nothing (except for handle_startendtag()):

**HTMLParser.**handle_starttag**(tag, attrs)**

This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of *(name, value)* pairs containing the attributes found inside the tag’s `<>` brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as handle_starttag('a', [('href', 'https://www.cwi.nl/')]).

All entity references from html.entities are replaced in the attribute values.

**HTMLParser.**handle_endtag**(tag)**

This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

**HTMLParser.**handle_startendtag**(tag, attrs)**

Similar to handle_starttag(), but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls handle_starttag() and handle_endtag().
HTMLParser. **handle_data**(data)

This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

HTMLParser. **handle_entityref**(name)

This method is called to process a named character reference of the form `&name;` (e.g. `&gt;`), where name is a general entity reference (e.g. 'gt'). This method is never called if `convert_charrefs` is True.

HTMLParser. **handle_charref**(name)

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `&gt;` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive '62' or 'x3E'. This method is never called if `convert_charrefs` is True.

HTMLParser. **handle_comment**(data)

This method is called when a comment is encountered (e.g. `<!----comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument 'comment'.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive '[if IE 9]>IE9-specific content<![endif]'.

HTMLParser. **handle_decl**(decl)

This method is called to handle an HTML doctype declaration (e.g.
<!DOCTYPE html>.

The `decl` parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `DOCTYPE html`).

**HTMLParser.** `handle_pi(data)`

Method called when a processing instruction is encountered. The `data` parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

**Note:** The **HTMLParser** class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing '?' will cause the '?' to be included in `data`.

**HTMLParser.** `unknown_decl(data)`

This method is called when an unrecognized declaration is read by the parser.

The `data` parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation does nothing.
Examples

The following class implements a parser that will be used to illustrate more examples:

```python
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("    attr:", attr)

    def handle_endtag(self, tag):
        print("End tag:", tag)

    def handle_data(self, data):
        print("Data:", data)

    def handle_comment(self, data):
        print("Comment:", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent:", c)

    def handle_decl(self, data):
```
print("Decl

:", data)

parser = MyHTMLParser()
Parsing a doctype:

>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
...
'"http://www.w3.org/TR/html4/strict.dtd">'
Decl
: DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "h

Parsing an element with a few attributes and a title:

>>> parser.feed('<img src="python-logo.png" alt="The Python l
Start tag: img
attr: ('src', 'python-logo.png')
attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data
: Python
End tag : h1

The content of script and style elements is returned as is, without
further parsing:

>>> parser.feed('<style type="text/css">#python { color: gree
Start tag: style
attr: ('type', 'text/css')
Data
: #python { color: green }
End tag : style

>>> parser.feed('<script type="text/javascript">'
...
'alert("<strong>hello!</strong>");</script>'
Start tag: script


attr: ('type', 'text/javascript')
Data : alert("<strong>hello!</strong>");
End tag : script

Parsing comments:

```python
>>> parser.feed('<!-- a comment -->')
... '<!--[if IE 9]>IE-specific content<![endif]-->
Comment : a comment
Comment : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to '>'):

```python
>>> parser.feed('&gt;&amp;#62;&amp;#x3E;')
Named ent : >
Num ent : >
Num ent : >
```

Feeding incomplete chunks to `feed()` works, but `handle_data()` might be called more than once (unless `convert_charrefs` is set to True):

```python
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', ...
... parser.feed(chunk)
... Start tag: span
Data : buff
Data : ered
Data : text
End tag : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:
```python
>>> parser.feed('<p><a class=link href=#main>tag soup</p>
Start tag: p
Start tag: a
    attr: ('class', 'link')
    attr: ('href', '#main')
Data : tag soup
End tag : p
End tag : a
```

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
html.entities — Definitions of HTML general entities

Source code: Lib/html/entities.py

This module defines four dictionaries, html5, name2codepoint, codepoint2name, and entitydefs.

html.entities.html5
A dictionary that maps HTML5 named character references [1] to the equivalent Unicode character(s), e.g. html5['gt;'] == '>'. Note that the trailing semicolon is included in the name (e.g. 'gt;'), however some of the names are accepted by the standard even without the semicolon: in this case the name is present with and without the ';'. See also html.unescape().

New in version 3.3.

html.entities.entitydefs
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

html.entities.name2codepoint
A dictionary that maps HTML entity names to the Unicode code points.

html.entities.codepoint2name
A dictionary that maps Unicode code points to HTML entity names.

Footnotes
XML Processing Modules

Source code: Lib/xml/ 

Python's interfaces for processing XML are grouped in the xml package.

Warning: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see the XML vulnerabilities and The defusedxml and defusedexpat Packages sections.

It is important to note that modules in the xml package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the xml.parsers.expat module will always be available.

The documentation for the xml.dom and xml.sax packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- xml.etree.ElementTree: the ElementTree API, a simple and lightweight XML processor
- xml.dom: the DOM API definition
- xml.dom.minidom: a minimal DOM implementation
- xml.dom.pulldom: support for building partial DOM trees
- xml.sax: SAX2 base classes and convenience functions
- xml.parsers.expat: the Expat parser binding
XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse XML features to carry out denial of service attacks, access local files, generate network connections to other machines, or circumvent firewalls.

The following table gives an overview of the known attacks and whether the various modules are vulnerable to them.

<table>
<thead>
<tr>
<th>kind</th>
<th>sax</th>
<th>etree</th>
<th>minidom</th>
<th>pulldom</th>
</tr>
</thead>
<tbody>
<tr>
<td>billion laughs</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
</tr>
<tr>
<td>quadratic blowup</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
<td>Vulnerable</td>
</tr>
<tr>
<td>external entity expansion</td>
<td>Safe (4)</td>
<td>Safe (1)</td>
<td>Safe (2)</td>
<td>Safe (4)</td>
</tr>
<tr>
<td>DTD retrieval</td>
<td>Safe (4)</td>
<td>Safe</td>
<td>Safe</td>
<td>Safe (4)</td>
</tr>
<tr>
<td>decompression bomb</td>
<td>Safe</td>
<td>Safe</td>
<td>Safe</td>
<td>Safe</td>
</tr>
</tbody>
</table>

1. `xml.etree.ElementTree` doesn’t expand external entities and raises a `ParserError` when an entity occurs.
2. `xml.dom.minidom` doesn’t expand external entities and simply returns the unexpanded entity verbatim.
3. `xmlrpclib` doesn’t expand external entities and omits them.
4. Since Python 3.7.1, external general entities are no longer processed by default.

billion laughs / exponential entity expansion

The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity
refers to another entity several times, and the final entity definition contains a small string. The exponential expansion results in several gigabytes of text and consumes lots of memory and CPU time.

quadratic blowup entity expansion
A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn’t as efficient as the exponential case but it avoids triggering parser countermeasures that forbid deeply-nested entities.

external entity expansion
Entity declarations can contain more than just text for replacement. They can also point to external resources or local files. The XML parser accesses the resource and embeds the content into the XML document.

DTD retrieval
Some XML libraries like Python’s xml.dom.pulldom retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

decompression bomb
Decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML streams such as gzipped HTTP streams or LZMA-compressed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation for defusedxml on PyPI has further information about all known attack vectors with examples and references.
The defusedxml and defusedexpat Packages

**defusedxml** is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. Use of this package is recommended for any server code that parses untrusted XML data. The package also ships with example exploits and extended documentation on more XML exploits such as XPath injection.

**defusedexpat** provides a modified libexpat and a patched pyexpat module that have countermeasures against entity expansion DoS attacks. The defusedexpat module still allows a sane and configurable amount of entity expansions. The modifications may be included in some future release of Python, but will not be included in any bugfix releases of Python because they break backward compatibility.
xml.etree.ElementTree — The ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py

The xml.etree.ElementTree module implements a simple and efficient API for parsing and creating XML data.

Changed in version 3.3: This module will use a fast implementation whenever available. The xml.etree.cElementTree module is deprecated.

Warning: The xml.etree.ElementTree module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.
Tutorial

This is a short tutorial for using `xml.etree.ElementTree` (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - `ElementTree` represents the whole XML document as a tree, and `Element` represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the `ElementTree` level. Interactions with a single XML element and its sub-elements are done on the `Element` level.

Parsing XML

We'll be using the following XML document as the sample data for this section:

```xml
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
    </country>
</data>
```
We can import this data by reading from a file:

```python
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```python
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an `Element`, which is the root element of the parsed tree. Other parsing functions may create an `ElementTree`. Check the documentation to be sure.

As an `Element`, `root` has a tag and a dictionary of attributes:

```python
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:
Children are nested, and we can access specific child nodes by index:

```python
>>> root[0][1].text
'2008'
```

**Note:** Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom `TreeBuilder` instance to the `XMLParser` constructor.

**Pull API for non-blocking parsing**

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an `XMLParser` and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed `Element` objects.
The most powerful tool for doing this is **XMLPullParser**. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with **XMLPullParser.feed()** calls. To get the parsed XML elements, call **XMLPullParser.read_events()**. Here is an example:

```python
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
... end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, **XMLPullParser** can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at **iterparse()**. It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

**Finding interesting elements**

**Element** has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For
example, `Element.iter()`:

```python
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

`Element.findall()` finds only elements with a tag which are direct children of the current element. `Element.find()` finds the first child with a particular tag, and `Element.text` accesses the element's text content. `Element.get()` accesses the element's attributes:

```python
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using XPath.

**Modifying an XML File**

`ElementTree` provides a simple way to build XML documents and write them to files. The `ElementTree.write()` method serves this purpose.
Once created, an `Element` object may be manipulated by directly changing its fields (such as `Element.text`), adding and modifying attributes (`Element.set()` method), as well as adding new children (for example with `Element.append()`).

Let's say we want to add one to each country's rank, and add an updated attribute to the rank element:

```python
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
... >>> tree.write('output.xml')
```

Our XML now looks like this:

```xml
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
```
We can remove elements using `Element.remove()`. Let’s say we want to remove all countries with a rank higher than 50:

```python
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...>>> tree.write('output.xml')
```

Our XML now looks like this:

```xml
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```
Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element:

```python
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /></a><c><d /></c>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the `prefix` is replaced by the full URI. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix “fictional” and the other serving as the default namespace:

```xml
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com" xmlns="http://people.example.com">
  <actor>
    <name>John Cleese</name>
    <fictional:character>Lancelot</fictional:character>
    <fictional:character>Archie Leach</fictional:character>
  </actor>
  <actor>
    <name>Eric Idle</name>
    <fictional:character>Sir Robin</fictional:character>
  </actor>
</actors>
```
One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a `find()` or `findall()`:

```python
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character')
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```python
ns = {'real_person': 'http://people.example.com',
     'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
```
Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.
XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the countrydata XML document from the Parsing XML section:

```python
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall("./year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall("./*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of
root.findall("./neighbor[2]")
```
## Supported XPath syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tag</code></td>
<td>Selects all child elements with the given tag. For example, <code>spam</code> selects all child elements named <code>spam</code>, and <code>spam/egg</code> selects all grandchildren named <code>egg</code> in all children named <code>spam</code>.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Selects all child elements. For example, <code>*/egg</code> selects all grandchildren named <code>egg</code>.</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.</td>
</tr>
<tr>
<td><code>//</code></td>
<td>Selects all subelements, on all levels beneath the current element. For example, <code>./egg</code> selects all <code>egg</code> elements in the entire tree.</td>
</tr>
<tr>
<td><code>..</code></td>
<td>Selects the parent element. Returns None if the path attempts to reach the ancestors of the start element (the element <code>find</code> was called on).</td>
</tr>
<tr>
<td><code>[@attrib]</code></td>
<td>Selects all elements that have the given attribute.</td>
</tr>
<tr>
<td><code>[@attrib='value']</code></td>
<td>Selects all elements for which the given attribute has the given value. The value cannot contain quotes.</td>
</tr>
<tr>
<td><code>[tag]</code></td>
<td>Selects all elements that have a child named <code>tag</code>. Only immediate children are supported.</td>
</tr>
<tr>
<td></td>
<td>Selects all elements whose complete text</td>
</tr>
<tr>
<td>Condition</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>.[text]</code></td>
<td>Selects all elements whose complete text content, including descendants, equals the given text.</td>
</tr>
<tr>
<td><code>tag='text'</code></td>
<td>Selects all elements that have a child named <code>tag</code> whose complete text content, including descendants, equals the given text.</td>
</tr>
<tr>
<td><code>[position]</code></td>
<td>Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (e.g. <code>last()-1</code>).</td>
</tr>
</tbody>
</table>

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.
Reference

Functions

```python
xml.etree.ElementTree.Comment(text=None)
```

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. `text` is a string containing the comment string. Returns an element instance representing a comment.

Note that `XMLParser` skips over comments in the input instead of creating comment objects for them. An `ElementTree` will only contain comment nodes if they have been inserted into the tree using one of the `Element` methods.

```python
xml.etree.ElementTree.dump(elem)
```

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

`elem` is an element tree or an individual element.

```python
xml.etree.ElementTree.fromstring(text, parser=None)
```

Parses an XML section from a string constant. Same as `XML()`. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

```python
xml.etree.ElementTree.fromstringlist(sequence,
```

...
Parses an XML document from a sequence of string fragments. sequence is a list or other sequence containing XML data fragments. parser is an optional parser instance. If not given, the standard XMLParser parser is used. Returns an Element instance.

New in version 3.2.

xml.etree.ElementTree.iselement(element)
Checks if an object appears to be a valid element object. element is an element instance. Returns a true value if this is an element object.

xml.etree.ElementTree.iterparse(source, events=None, parser=None)
 Parses an XML section into an element tree incrementally, and reports what's going on to the user. source is a filename or file object containing XML data. events is a sequence of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the “ns” events are used to get detailed namespace information). If events is omitted, only "end" events are reported. parser is an optional parser instance. If not given, the standard XMLParser parser is used. parser must be a subclass of XMLParser and can only use the default TreeBuilder as a target. Returns an iterator providing (event, elem) pairs.

Note that while iterparse() builds the tree incrementally, it issues blocking reads on source (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see XMLPullParser.
**Note:** `iterparse()` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present. If you need a fully populated element, look for “end” events instead.

*Deprecated since version 3.4: The parser argument.*

```python
xml.etree.ElementTree.parse(source, parser=None)
```

Parses an XML section into an element tree. `source` is a filename or file object containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

```python
xml.etree.ElementTree.ProcessingInstruction(target, text=None)
```

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. `target` is a string containing the PI target. `text` is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that `XMLParser` skips over processing instructions in the input instead of creating comment objects for them. An `ElementTree` will only contain processing instruction nodes if they have been inserted into to the tree using one of the `Element` methods.

```python
xml.etree.ElementTree.register_namespace(prefix, uri)
```

Registers a namespace prefix. The registry is global, and any
existing mapping for either the given prefix or the namespace URI will be removed. \textit{prefix} is a namespace prefix. \textit{uri} is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

\textit{New in version 3.2.}

\texttt{xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)}

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. \textit{parent} is the parent element. \textit{tag} is the subelement name. \textit{attrib} is an optional dictionary, containing element attributes. \textit{extra} contains additional attributes, given as keyword arguments. Returns an element instance.

\texttt{xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml", *, short_empty_elements=True)}

Generates a string representation of an XML element, including all subelements. \textit{element} is an \texttt{Element} instance. \textit{encoding} \footnote{1} is the output encoding (default is US-ASCII). Use \texttt{encoding="unicode"} to generate a Unicode string (otherwise, a bytestring is generated). \textit{method} is either "xml", "html" or "text" (default is "xml"). \textit{short_empty_elements} has the same meaning as in \texttt{ElementTree.write()}. Returns an (optionally) encoded string containing the XML data.

\textit{New in version 3.4: The short_empty_elements parameter.}

\texttt{xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml", *, short_empty_elements=True)}
Generates a string representation of an XML element, including all subelements. `element` is an `Element` instance. `encoding` [1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). `method` is either "xml", "html" or "text" (default is "xml"). `short_empty_elements` has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that 
\[
\text{b"".join(tostringlist(element))} == \text{tostring(element)}.
\]

*New in version 3.2.*

*New in version 3.4:* The `short_empty_elements` parameter.

```python
xml.etree.ElementTree.XML(text, parser=None)
```

Parses an XML section from a string constant. This function can be used to embed “XML literals” in Python code. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

```python
xml.etree.ElementTree.XMLID(text, parser=None)
```

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

**Element Objects**
`xml.etree.ElementTree.Element(tag, attrib={}, **extra)`

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. `tag` is the element name. `attrib` is an optional dictionary, containing element attributes. `extra` contains additional attributes, given as keyword arguments.

**tag**

A string identifying what kind of data this element represents (the element type, in other words).

**text**

**tail**

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the `text` attribute holds either the text between the element’s start tag and its first child or end tag, or `None`, and the `tail` attribute holds either the text between the element’s end tag and the next tag, or `None`. For the XML data

```xml
<a><b>1</c><d/3</c></b>4</a>
```

the `a` element has `None` for both `text` and `tail` attributes, the `b` element has `text "1"` and `tail "4"`, the `c` element has `text "2"` and `tail None`, and the `d` element has `text None` and `tail "3"`.

To collect the inner text of an element, see `itertext()`, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.
attrib
A dictionary containing the element’s attributes. Note that while the attrib value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()
Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to None.

get(key, default=None)
Gets the element attribute named key.

Returns the attribute value, or default if the attribute was not found.

items()
Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()
Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(key, value)
Set the attribute key on the element to value.
The following methods work on the element’s children (subelements).

**append**(subelement)

Adds the element *subelement* to the end of this element’s internal list of subelements. Raises *TypeError* if *subelement* is not an *Element*.

**extend**(subelements)

Appends *subelements* from a sequence object with zero or more elements. Raises *TypeError* if a subelement is not an *Element*.

*New in version 3.2.*

**find**(match, namespaces=None)

Finds the first subelement matching *match*. *match* may be a tag name or a path. Returns an element instance or None. *namespaces* is an optional mapping from namespace prefix to full name.

**findall**(match, namespaces=None)

Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

**findtext**(match, default=None, namespaces=None)

Finds text for the first subelement matching *match*. *match* may be a tag name or a path. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is
returned. *namespaces* is an optional mapping from namespace prefix to full name.

**getchildren()**

*Deprecated since version 3.2: Use* `list(elem)` *or iteration.*

**getiterator**(tag=None)

*Deprecated since version 3.2: Use method* `Element.iter()` *instead.*

**insert**(index, subelement)

Inserts *subelement* at the given position in this element. Raises *TypeError* if *subelement* is not an *Element*.

**iter**(tag=None)

Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not *None* or '*'* , only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

*New in version 3.2.*

**iterfind**(match, namespaces=None)

Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

*New in version 3.2.*

**itertext()**
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

*New in version 3.2.*

**makeelement**(*tag, attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the **SubElement()** factory function instead.

**remove**(subelement)

Removes subelement from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

**Element** objects also support the following sequence type methods for working with subelements: **__delitem__()**, **__getitem__()**, **__setitem__()**, **__len__()**.

Caution: Elements with no subelements will test as False. This behavior will change in future versions. Use specific len(elem) or elem is None test instead.

```python
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no s")

if element is None:
    print("element not found")
```

**ElementTree** Objects
class xml.etree.ElementTree.**ElementTree**(element=None, file=None)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

*element* is the root element. The tree is initialized with the contents of the XML *file* if given.

**_setroot(element)**

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

**find(match, namespaces=None)**

Same as `Element.find()`, starting at the root of the tree.

**findall(match, namespaces=None)**

Same as `Element.findall()`, starting at the root of the tree.

**findtext(match, default=None, namespaces=None)**

Same as `Element.findtext()`, starting at the root of the tree.

**getiterator(tag=None)**

Deprecation since version 3.2: Use method `ElementTree.iter()` instead.

**getroot()**

Returns the root element for this tree.

**iter(tag=None)**

Creates and returns a tree iterator for the root element. The
iterator loops over all elements in this tree, in section order. tag is
the tag to look for (default is to return all elements).

**iterfind**(*match, namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

*New in version 3.2.*

**parse**(*source, parser=None*)

Loads an external XML section into this element tree. source is a
file name or file object. parser is an optional parser instance. If
not given, the standard `XMLParser` parser is used. Returns the
section root element.

**write**(*file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml", *, short_empty_elements=True*)

Writes the element tree to a file, as XML. file is a file name, or a
file object opened for writing. encoding [1] is the output
encoding (default is US-ASCII). xml_declaration controls if an
XML declaration should be added to the file. Use False for
never, True for always, None for only if not US-ASCII or UTF-8
or Unicode (default is None). default_namespace sets the
default XML namespace (for “xmlns”). method is either "xml",
"html" or "text" (default is "xml"). The keyword-only
short_empty_elements parameter controls the formatting of
elements that contain no content. If True (the default), they are
emitted as a single self-closed tag, otherwise they are emitted
as a pair of start/end tags.

The output is either a string (str) or binary (bytes). This is
controlled by the encoding argument. If encoding is "unicode", the output is a string; otherwise, it's binary. Note that this may conflict with the type of file if it's an open file object; make sure you do not try to write a string to a binary stream and vice versa.

New in version 3.4: The short_empty_elements parameter.

This is the XML file that is going to be manipulated:

```xml
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example or <a href="http://example.com/">example.com</a></p>
  </body>
</html>
```

Example of changing the attribute “target” of every link in first paragraph:

```python
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")  # Finds first occurrence of 'p'
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))  # Returns list of all 'a' elements
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77e8c>]
>>> for i in links:  # Iterates through all links
...    i.attrib["target"] = "blank"
```
QName Objects

```python
tree.write("output.xhtml")
```

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. `text_or_uri` is a string containing the QName value, in the form `{uri}local`, or, if the tag argument is given, the URI part of a QName. If `tag` is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. QName instances are opaque.

TreeBuilder Objects

```python
class xml.etree.ElementTree.TreeBuilder(element_factory=None)
```

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. `element_factory`, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

```python
close()
```

Flushes the builder buffers, and returns the toplevel document element. Returns an Element instance.

```python
data(data)
```

Adds text to the current element. `data` is a string. This should be
either a bytestring, or a Unicode string.

end(tag)
Closes the current element. tag is the element name. Returns the closed element.

start(tag, attrs)
Opens a new element. tag is the element name. attrs is a dictionary containing element attributes. Returns the opened element.

In addition, a custom TreeBuilder object can provide the following method:

doctype(name, pubid, system)
Handles a doctype declaration. name is the doctype name. pubid is the public identifier. system is the system identifier. This method does not exist on the default TreeBuilder class.

New in version 3.2.

XMLParser Objects

class xml.etree.ElementTree.XMLParser(html=0, target=None, encoding=None)
This class is the low-level building block of the module. It uses xml.parsers.expat for efficient, event-based parsing of XML. It can be fed XML data incrementally with the feed() method, and parsing events are translated to a push API - by invoking callbacks on the target object. If target is omitted, the standard TreeBuilder is used. The html argument was historically used for backwards compatibility and is now deprecated. If encoding [1]
is given, the value overrides the encoding specified in the XML file.

*Deprecated since version 3.4:* The `html` argument. The remaining arguments should be passed via keyword to prepare for the removal of the `html` argument.

**close()**
Finishes feeding data to the parser. Returns the result of calling the `close()` method of the `target` passed during construction; by default, this is the toplevel document element.

**doctype**(name, pubid, system)
*Deprecated since version 3.2:* Define the `TreeBuilder.doctype()` method on a custom TreeBuilder target.

**feed**(data)
Feeds data to the parser. `data` is encoded data.

`XMLParser.feed()` calls `target`'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. `XMLParser.close()` calls `target`'s method `close()`.

`XMLParser` can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```python
>>> from xml.etree.ElementTree import XMLParser

>>> class MaxDepth:
...     # The target object of the parser...
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):
...             # Called
...             self.depth += 1
...             if self.depth > self.maxDepth:
```
XMLPullParser Objects

class xml.etree.ElementTree/XMLPullParser(events=None)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of XMLParser, but instead of pushing calls to a callback target, XMLPullParser collects an internal list of parsing events and lets the user read from it. events is a sequence of events to report back. The supported events are the strings

```python
... self.maxDepth = self.depth
... def end(self, tag): # Called
... self.depth -= 1
... def data(self, data):
... pass # We do not need to do anything
... def close(self): # Called when all data has been read
... return self.maxDepth

>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """"
... <a>
... <b>
... </b>
... <b>
... <c>
... <d>
... </d>
... </c>
... </b>
... </a>"
>>> parser.feed(exampleXml)
>>> parser.close()
4
```
"start", "end", "start-ns" and "end-ns" (the "ns" events are used to get detailed namespace information). If events is omitted, only "end" events are reported.

**feed**(*data*)

Feed the given bytes data to the parser.

**close**()

Signal the parser that the data stream is terminated. Unlike `XMLParser.close()`, this method always returns `None`. Any events not yet retrieved when the parser is closed can still be read with `read_events()`.

**read_events**()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields *(event, elem)* pairs, where *event* is a string representing the type of event (e.g. "end") and *elem* is the encountered *Element* object.

Events provided in a previous call to `read_events()` will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from `read_events()` will have unpredictable results.

**Note:** `XMLPullParser` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for “end” events
Exceptions

```python
class xml.etree.ElementTree.ParseError
```

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

**code**
A numeric error code from the expat parser. See the documentation of `xml.parsers.expat` for the list of error codes and their meanings.

**position**
A tuple of *line, column* numbers, specifying where the error occurred.

Footnotes

(1, 2, 3, 4) The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.
The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program’s position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or “levels” in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.
DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a DOMImplementation object class which provides access to Document creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing Document object. In Python, each DOM implementation will provide a function `getDOMImplementation()`. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section Conformance for a detailed discussion of mapping requirements.

See also:

**Document Object Model (DOM) Level 2 Specification**

The W3C recommendation upon which the Python DOM API is based.

**Document Object Model (DOM) Level 1 Specification**
The W3C recommendation for the DOM supported by `xml.dom.minidom`.

**Python Language Mapping Specification**
This specifies the mapping from OMG IDL to Python.
Module Contents

The `xml.dom` contains the following functions:

```python
def registerDOMImplementation(name, factory):
    Register the `factory` function with the name `name`. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

def getDOMImplementation(name=None, features=()):
    Return a suitable DOM implementation. The `name` is either well-known, the module name of a DOM implementation, or `None`. If it is not `None`, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

    If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an `ImportError`. The features list must be a sequence of `(feature, version)` pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

```
```

```python
`EMPTY_NAMESPACE`

    The value used to indicate that no namespace is associated with a
node in the DOM. This is typically found as the namespaceURI of a node, or used as the namespaceURI parameter to a namespaces-specific method.

```python
xml.dom.XML_NAMESPACE
```

The namespace URI associated with the reserved prefix `xml`, as defined by [Namespaces in XML](https://www.w3.org/TR/2006/REC-xml-names-20060801/) (section 4).

```python
xml.dom.XMLNS_NAMESPACE
```

The namespace URI for namespace declarations, as defined by [Document Object Model (DOM) Level 2 Core Specification](https://www.w3.org/TR/2006/REC-dom-Level-2-Core-Specified-Features-20060801/) (section 1.1.8).

```python
xml.dom.XHTML_NAMESPACE
```

The URI of the XHTML namespace as defined by [XHTML 1.0: The Extensible HyperText Markup Language](https://www.w3.org/TR/xhtml1/) (section 3.1.1).

In addition, `xml.dom` contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the `nodeType` attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.
Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMImplementation</td>
<td>DOMImplementation Objects</td>
<td>Interface to the underlying implementation</td>
</tr>
<tr>
<td>Node</td>
<td>Node Objects</td>
<td>Base interface for most objects in a document</td>
</tr>
<tr>
<td>NodeList</td>
<td>NodeList Objects</td>
<td>Interface for a sequence of nodes.</td>
</tr>
<tr>
<td>DocumentType</td>
<td>DocumentType Objects</td>
<td>Information about the declarations needed to process a document.</td>
</tr>
<tr>
<td>Document</td>
<td>Document Objects</td>
<td>Object which represents an entire document</td>
</tr>
<tr>
<td>Element</td>
<td>Element Objects</td>
<td>Element nodes in the document</td>
</tr>
</tbody>
</table>
An additional section describes the exceptions defined for working with the DOM in Python.

**DOMImplementation Objects**

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

```python
DOMImplementation.hasFeature(feature, version)
```

Return true if the feature identified by the pair of strings `feature` and `version` is implemented.

```python
DOMImplementation.createDocument(namespaceUri,
```
qualifiedName, doctype)
Return a new Document object (the root of the DOM), with a child
Element object having the given namespaceUri and
qualifiedName. The doctype must be a DocumentType object
created by createDocumentType(), or None. In the Python
DOM API, the first two arguments can also be None in order to
indicate that no Element child is to be created.

DOMImplementation.createDocumentType(qualifiedName,
publicId, systemId)
Return a new DocumentType object that encapsulates the given
qualifiedName, publicId, and systemId strings, representing the
information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of Node.

Node.nodeType
An integer representing the node type. Symbolic constants for the
types are on the Node object: ELEMENT_NODE,
ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE,
ENTITY_NODE, PROCESSING_INSTRUCTION_NODE,
COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE,
NOTATION_NODE. This is a read-only attribute.

Node.parentNode
The parent of the current node, or None for the document node.
The value is always a Node object or None. For Element nodes,
this will be the parent element, except for the root element, in
which case it will be the Document object. For Attr nodes, this is
always None. This is a read-only attribute.

Node. **attributes**
A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

Node. **previousSibling**
The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the self element’s start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

Node. **nextSibling**
The node that immediately follows this one with the same parent. See also previousSibling. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node. **childNodes**
A list of nodes contained within this node. This is a read-only attribute.

Node. **firstChild**
The first child of the node, if there are any, or None. This is a read-only attribute.

Node. **lastChild**
The last child of the node, if there are any, or None. This is a read-only attribute.

Node. **localName**
The part of the `tagName` following the colon if there is one, else the entire `tagName`. The value is a string.

**Node. `prefix`**

The part of the `tagName` preceding the colon if there is one, else the empty string. The value is a string, or `None`.

**Node. `namespaceURI`**

The namespace associated with the element name. This will be a string or `None`. This is a read-only attribute.

**Node. `nodeName`**

This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the `tagName` property for elements or the `name` property for attributes. For all node types, the value of this attribute will be either a string or `None`. This is a read-only attribute.

**Node. `nodeValue`**

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with `nodeName`. The value is a string or `None`.

**Node. `hasAttributes()`**

Returns true if the node has any attributes.

**Node. `hasChildNodes()`**

Returns true if the node has any child nodes.

**Node. `isSameNode(other)`**

Returns true if `other` refers to the same node as this node. This is
especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

**Note:** This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

**Node.** `appendChild(newChild)`

Add a new child node to this node at the end of the list of children, returning `newChild`. If the node was already in the tree, it is removed first.

**Node.** `insertBefore(newChild, refChild)`

Insert a new child node before an existing child. It must be the case that `refChild` is a child of this node; if not, `ValueError` is raised. `newChild` is returned. If `refChild` is `None`, it inserts `newChild` at the end of the children's list.

**Node.** `removeChild(oldChild)`

Remove a child node. `oldChild` must be a child of this node; if not, `ValueError` is raised. `oldChild` is returned on success. If `oldChild` will not be used further, its `unlink()` method should be called.

**Node.** `replaceChild(newChild, oldChild)`

Replace an existing node with a new node. It must be the case that `oldChild` is a child of this node; if not, `ValueError` is raised.

**Node.** `normalize()`

Join adjacent text nodes so that all stretches of text are stored as
single Text instances. This simplifies processing text from a DOM tree for many applications.

**Node.cloneNode** (*deep*)

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

**NodeList Objects**

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

**NodeList.item**(i)

Return the *i*’th item from the sequence, if there is one, or **None**. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

**NodeList.length**

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for **__len__()** and **__getitem__()**; this allows iteration over the NodeList in **for** statements and proper support for the **len()** built-
If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

**DocumentType Objects**

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

**DocumentType.** publicId

The public identifier for the external subset of the document type definition. This will be a string or None.

**DocumentType.** systemId

The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

**DocumentType.** internalSubset

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.
DocumentType.name
The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType.entities
This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType.notations
This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

Document Objects
A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

Document.documentElement
The one and only root element of the document.

Document.createElement(tagName)
Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().
Document. `createElementNS(namespaceURI, tagName)`
Create and return a new element with a namespace. The `tagName` may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

Document. `createTextNode(data)`
Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document. `createComment(data)`
Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document. `createProcessingInstruction(target, data)`
Create and return a processing instruction node containing the `target` and `data` passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document. `createAttribute(name)`
Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate Element object to use the newly created attribute instance.

Document. `createAttributeNS(namespaceURI, qualifiedName)`
Create and return an attribute node with a namespace. The `tagName` may have a prefix. This method does not associate the attribute node with any particular element. You must use
setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document. **getElementsByTagName(tagName)**
Search for all descendants (direct children, children’s children, etc.) with a particular element type name.

Document. **getElementsByTagNameNS(namespaceURI, localName)**
Search for all descendants (direct children, children’s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

**Element Objects**

Element is a subclass of Node, so inherits all the attributes of that class.

**Element. **tagName**
The element type name. In a namespace-using document it may have colons in it. The value is a string.

**Element. **getElementsByTagName(tagName)**
Same as equivalent method in the Document class.

**Element. **getElementsByTagNameNS(namespaceURI, localName)**
Same as equivalent method in the Document class.

**Element. **hasAttribute(name)**
Returns true if the element has an attribute named by name.

**Element. **hasAttributeNS(namespaceURI, localName)**
Returns true if the element has an attribute named by namespaceURI and localName.

**Element.getAttribute(name)**
Return the value of the attribute named by name as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

**Element.getAttributeNode(attrname)**
Return the Attr node for the attribute named by attrname.

**Element.getAttributeNS(namespaceURI, localName)**
Return the value of the attribute named by namespaceURI and localName as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

**Element.getAttributeNodeNS(namespaceURI, localName)**
Return an attribute value as a node, given a namespaceURI and localName.

**Element.removeAttribute(name)**
Remove an attribute by name. If there is no matching attribute, a NotFoundErr is raised.

**Element.removeAttributeNode(oldAttr)**
Remove and return oldAttr from the attribute list, if present. If oldAttr is not present, NotFoundErr is raised.

**Element.removeAttributeNS(namespaceURI, localName)**
Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

**Element.setAttribute(name, value)**
Set an attribute value from a string.

**Element**. `.setAttributeNode(newAttr)`

Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If `newAttr` is already in use, `InuseAttributeErr` will be raised.

**Element**. `.setAttributeNodeNS(newAttr)`

Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and `localName` attributes match. If a replacement occurs, the old attribute node will be returned. If `newAttr` is already in use, `InuseAttributeErr` will be raised.

**Element**. `.setAttributeNS(namespaceURI, qname, value)`

Set an attribute value from a string, given a `namespaceURI` and a `qname`. Note that a qname is the whole attribute name. This is different than above.

**Attr Objects**

`Attr` inherits from `Node`, so inherits all its attributes.

**Attr**. `.name`

The attribute name. In a namespace-using document it may include a colon.

**Attr**. `.localName`

The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.
**Attr. `prefix`**

The part of the name preceding the colon if there is one, else the empty string.

**Attr. `value`**

The text value of the attribute. This is a synonym for the `nodeValue` attribute.

---

**NamedNodeMap Objects**

NamedNodeMap does not inherit from Node.

**NamedNodeMap. `length`**

The length of the attribute list.

**NamedNodeMap. `item`**

The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the `value` attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized `getAttribute*()` family of methods on the Element objects.

---

**Comment Objects**

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

**Comment. `data`**

The content of the comment as a string. The attribute contains all
characters between the leading `<!-` and trailing `-->`, but does not include them.

Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

Text.data

The content of the text node as a string.

**Note:** The use of a CDATASection node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent CDATASection nodes represent different CDATA marked sections.

ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.

ProcessingInstruction.target

The content of the processing instruction up to the first whitespace
character. This is a read-only attribute.

ProcessingInstruction.data

The content of the processing instruction following the first whitespace character.

Exceptions

The DOM Level 2 recommendation defines a single exception, DOMException, and a number of constants that allow applications to determine what sort of error occurred. DOMException instances carry a code attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the code attribute.

exception xml.dom.DOMException

Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception xml.dom.DomStringSizeErr

Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception xml.dom.HierarchyRequestErr

Raised when an attempt is made to insert a node where the node type is not allowed.
**exception xml.dom.IndexSizeErr**
Raised when an index or size parameter to a method is negative or exceeds the allowed values.

**exception xml.dom.InuseAttributeErr**
Raised when an attempt is made to insert an Attr node that is already present elsewhere in the document.

**exception xml.dom.InvalidAccessErr**
Raised if a parameter or an operation is not supported on the underlying object.

**exception xml.dom.InvalidCharacterErr**
This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an Element node with a space in the element type name will cause this error to be raised.

**exception xml.dom.InvalidModificationErr**
Raised when an attempt is made to modify the type of a node.

**exception xml.dom.InvalidStateErr**
Raised when an attempt is made to use an object that is not defined or is no longer usable.

**exception xml.dom.NamespaceErr**
If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

**exception xml.dom.NotFoundErr**
Exception when a node does not exist in the referenced context.
For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

```plaintext
exception xml.dom.NotSupportedException
Raised when the implementation does not support the requested type of object or operation.
```

```plaintext
exception xml.dom.NoDataAllowedErr
This is raised if data is specified for a node which does not support data.
```

```plaintext
exception xml.dom.NoModificationAllowedErr
Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).
```

```plaintext
exception xml.dom.SyntaxErr
Raised when an invalid or illegal string is specified.
```

```plaintext
exception xml.dom.WrongDocumentErr
Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.
```

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSTRING_SIZE_ERR</td>
<td>DomstringSizeErr</td>
</tr>
<tr>
<td>HIERARCHY_REQUEST_ERR</td>
<td>HierarchyRequestErr</td>
</tr>
<tr>
<td>INDEX_SIZE_ERR</td>
<td>IndexSizeErr</td>
</tr>
<tr>
<td>INUSE_ATTRIBUTE_ERR</td>
<td>InuseAttributeErr</td>
</tr>
<tr>
<td>INVALID_ACCESS_ERR</td>
<td>InvalidAccessErr</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Name</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>INVALID_CHARACTER_ERR</td>
<td>InvalidCharacterErr</td>
</tr>
<tr>
<td>INVALID_MODIFICATION_ERR</td>
<td>InvalidModificationErr</td>
</tr>
<tr>
<td>INVALID_STATE_ERR</td>
<td>InvalidStateErr</td>
</tr>
<tr>
<td>NAMESPACE_ERR</td>
<td>NamespaceErr</td>
</tr>
<tr>
<td>NOT_FOUND_ERR</td>
<td>NotFoundErr</td>
</tr>
<tr>
<td>NOT_SUPPORTED_ERR</td>
<td>NotSupportedErr</td>
</tr>
<tr>
<td>NO_DATA_ALLOWED_ERR</td>
<td>NoDataAllowedErr</td>
</tr>
<tr>
<td>NO_MODIFICATION_ALLOWED_ERR</td>
<td>NoModificationAllowedErr</td>
</tr>
<tr>
<td>SYNTAX_ERR</td>
<td>SyntaxErr</td>
</tr>
<tr>
<td>WRONG_DOCUMENT_ERR</td>
<td>WrongDocumentErr</td>
</tr>
</tbody>
</table>
Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool or int</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>long int</td>
<td>int</td>
</tr>
<tr>
<td>unsigned int</td>
<td>int</td>
</tr>
<tr>
<td>DOMString</td>
<td>str or bytes</td>
</tr>
<tr>
<td>null</td>
<td>None</td>
</tr>
</tbody>
</table>

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL attribute declarations in much the way the Java mapping does. Mapping the IDL declarations

```python
readonly attribute string someValue;
attribute string anotherValue;
```

yields three accessor functions: a “get” method for someValue
(_get_someValue()), and “get” and “set” methods for anotherValue (_get_anotherValue() and _set_anotherValue()). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: object.someValue is not required to work, and may raise an AttributeError.

The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared readonly may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. “Set” accessors should never be provided for readonly attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of getElementsByTagName(), being “live”. The Python DOM API does not require implementations to enforce such requirements.
**xml.dom.minidom** — Minimal DOM implementation

**Source code:** [Lib/xml/dom/minidom.py](https://docs.python.org/3/library/xml.dom.minidom.html)

`xml.dom.minidom` is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the `xml.etree.ElementTree` module for their XML processing instead.

**Warning:** The `xml.dom.minidom` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](https://en.wikipedia.org/wiki/XML_vulnerability).

DOM applications typically start by parsing some XML into a DOM. With `xml.dom.minidom`, this is done through the parse functions:

```python
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\temp\mydata.xml')  # parse an XML file

datasource = open('c:\temp\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>
```

The `parse()` function can take either a filename or an open file.
xml.dom.minidom.**parse**(filename_or_file, parser=None, bufsize=None)

Return a Document from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the **parseString()** function instead:

xml.dom.minidom.**parseString**(string, parser=None)

Return a Document that represents the *string*. This method creates an `io.StringIO` object for the string and passes that on to **parse()**.

Both functions return a Document object representing the content of the document.

What the **parse()** and **parseString()** functions do is connect an XML parser with a “DOM builder” that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it’s simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a “DOM Implementation” object. You can get this object either by calling the **getDOMImplementation()** function in the `xml.dom` package or the
xml.dom.minidom module. Once you have a Document, you can add child nodes to it to populate the DOM:

```python
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```python
dom3 = parseString("<myxml>Some data</myxml>"
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an xml.dom.minidom-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python’s garbage collector will eventually take care of the objects in the tree.

**See also:**

**Document Object Model (DOM) Level 1 Specification**

The W3C recommendation for the DOM supported by
xml.dom.minidom.
DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

Node. `unlink()`

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the Document object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically unlink `dom` when the with block is exited:

```python
with xml.dom.minidom.parse(datasource) as dom:
    ...
    # Work with dom.
```

Node. `writexml(writer, indent='', addindent='', newl='')`

Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface. The `indent` parameter is the indentation of the current node. The `addindent` parameter is the incremental indentation to use for subnodes of the current one. The `newl` parameter specifies the string to use to terminate newlines.

For the Document node, an additional keyword argument
encoding can be used to specify the encoding field of the XML header.

Node. `toxml(encoding=None)`
Return a string or byte string containing the XML represented by the DOM node.

With an explicit encoding [1] argument, the result is a byte string in the specified encoding. With no encoding argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

Node. `toprettyxml(indent="\t", newl="\n", encoding=None)`
Return a pretty-printed version of the document. indent specifies the indentation string and defaults to a tabulator; newl specifies the string emitted at the end of each line and defaults to \n.

The encoding argument behaves like the corresponding argument of `toxml()`.
DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```python
import xml.dom.minidom

document = ""
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
  <point>This is a demo</point>
  <point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
  <point>It is important</point>
  <point>To have more than</point>
  <point>one slide</point>
</slide>
</slideshow>
""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
```
def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title"))
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>"
          for point in points:
            handlePoint(point)
        print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshowTitle(slideshow.getElementsByTagName("slide")
slides = slideshow.getElementsByTagName("slide")
handleToc(slides)
handleSlides(slides)
print("</html>")

handleSlideshow(dom)
minidom and the DOM standard

The `xml.dom.minidom` module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the Document object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. `xml.dom.minidom` supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`).
they must not be changed.

- **DOMException** is currently not supported in `xml.dom.minidom`. Instead, `xml.dom.minidom` uses standard Python exceptions such as `TypeError` and `AttributeError`.
- NodeList objects are implemented using Python’s built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more “Pythonic” than the interface defined in the W3C recommendations.

The following interfaces have no implementation in `xml.dom.minidom`:

- DOMTimeStamp
- DocumentType
- DOMImplementation
- CharacterData
- CDATASection
- Notation
- Entity
- EntityReference
- DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.

**Footnotes**

The encoding name included in the XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not valid in an XML document’s declaration, even though Python accepts it as an encoding name. See [https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl](https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl) and [https://www.iana.org/assignments/character-](https://www.iana.org/assignments/character-
xml.dom.pulldom — Support for building partial DOM trees

Source code: Lib/xml/dom/pulldom.py

The xml.dom.pulldom module provides a “pull parser” which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling “events” from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

Warning: The xml.dom.pulldom module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Changed in version 3.7.1: The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in:

```python
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```
Example:

```python
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName:
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event is a constant and can be one of:

- START_ELEMENT
- END_ELEMENT
- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNOREABLE_WHITESPACE

node is an object of type xml.dom.minidom.Document, xml.dom.minidom.Element or xml.dom.minidom.Text.

Since the document is treated as a “flat” stream of events, the document “tree” is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the DOMEventStream.expandNode()
method and switch to DOM-related processing.

```python
class xml.dom.pulldom.PullDom(documentFactory=None)
    Subclass of xml.sax.handler.ContentHandler.

class xml.dom.pulldom.SAX2DOM(documentFactory=None)
    Subclass of xml.sax.handler.ContentHandler.
```

```python
xml.dom.pulldom.parse(stream_or_string, parser=None, bufsize=None)
    Return a DOMEventStream from the given input. stream_or_string may be either a file name, or a file-like object. parser, if given, must be an XMLReader object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the parseString() function instead:

```python
xml.dom.pulldom.parseString(string, parser=None)
    Return a DOMEventStream that represents the (Unicode) string.
```

```python
xml.dom.pulldom.default_bufsize
    Default value for the bufsize parameter to parse().

    The value of this variable can be changed before calling parse() and the new value will take effect.
```
DOMEventStream Objects

class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)

getEvent()
Return a tuple containing event and the current node as xml.dom.minidom.Document if event equals START_DOCUMENT, xml.dom.minidom.Element if event equals START_ELEMENT or END_ELEMENT or xml.dom.minidom.Text if event equals CHARACTERS. The current node does not contain information about its children, unless expandNode() is called.

expandNode(node)
Expands all children of node into node. Example:

```python
from xml.dom import pulldom

dom = pulldom.parseString(xml)
for event, node in dom:
    if event == pulldom.START_ELEMENT and node:
        # Following statement only prints '<p/>
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with
        print(node.toxml())
```

reset()
xml.sax — Support for SAX2 parsers

Source code: Lib/xml/sax/__init__.py

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

**Warning:** The xml.sax module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

*Changed in version 3.7.1:* The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method `setFeature()` on the parser object and argument `feature_external_ges`.

The convenience functions are:

```python
xml.sax.make_parser(poster_list=[])  # Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is provided, it must be a list of strings which name modules that have a function named create_parser(). Modules listed in parser_list will be used before modules in the default list of parsers.
```
`xml.sax.parse(filename_or_stream, handler, error_handler=handler.ErrorHandler())`

Create a SAX parser and use it to parse a document. The document, passed in as `filename_or_stream`, can be a filename or a file object. The `handler` parameter needs to be a SAX `ContentHandler` instance. If `error_handler` is given, it must be a SAX `ErrorHandler` instance; if omitted, `SAXParseError` will be raised on all errors. There is no return value; all work must be done by the `handler` passed in.

`xml.sax.parseString(string, handler, error_handler=handler.ErrorHandler())`

Similar to `parse()`, but parses from a buffer `string` received as a parameter. `string` must be a `str` instance or a bytes-like object.

*Changed in version 3.5:* Added support of `str` instances.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. “Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes,
but applications may use implementations which do not inherit from the provided classes. The InputSource, Locator, Attributes, AttributesNS, and XMLReader interfaces are defined in the module xml.sax.xmlreader. The handler interfaces are defined in xml.sax.handler. For convenience, InputSource (which is often instantiated directly) and the handler classes are also available from xml.sax. These interfaces are described below.

In addition to these classes, xml.sax provides the following exception classes.

```
exception xml.sax.SAXException(msg, exception=None)
```

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, msg should be a human-readable description of the error. The optional exception parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

```
exception xml.sax.SAXParseException(msg, exception, locator)
```

Subclass of SAXException raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class
supports the SAX **Locator** interface as well as the **SAXException** interface.

```
exception xml.sax.SAXNotRecognizedException(msg, exception=None)
```

Subclass of **SAXException** raised when a SAX **XMLReader** is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

```
exception xml.sax.SAXNotSupportedException(msg, exception=None)
```

Subclass of **SAXException** raised when a SAX **XMLReader** is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

**See also:**

**SAX: The Simple API for XML**

This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

**Module** `xml.sax.handler`

Definitions of the interfaces for application-provided objects.

**Module** `xml.sax.saxutils`

Convenience functions for use in SAX applications.

**Module** `xml.sax.xmlreader`

Definitions of the interfaces for parser-provided objects.
SAXException Objects

The `SAXException` exception class supports the following methods:

**SAXException**. `getMessage()`  
Return a human-readable message describing the error condition.

**SAXException**. `getException()`  
Return an encapsulated exception object, or `None`. 
The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module `xml.sax.handler`, so that all methods get default implementations.

```
class xml.sax.handler.ContentHandler
    This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.DTDHandler
    Handle DTD events.

    This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.EntityResolver
    Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.
```
class xml.sax.handler.ErrorHandler

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, xml.sax.handler provides symbolic constants for the feature and property names.

xml.sax.handler.feature_namespaces

value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_namespace_prefixes

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_string_interning

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be
access: (parsing) read-only; (not parsing) read/write

**xml.sax.handler.feature_validation**

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

**xml.sax.handler.feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

**xml.sax.handler.feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

**xml.sax.handler.all_features**
List of all features.

**xml.sax.handler.property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.property_declaration_handler

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.property_dom_node

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.property_xml_string

value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.all_properties

List of all known property names.
ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

`ContentHandler.setDocumentLocator(locator)`

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application’s business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

`ContentHandler.startDocument()`

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for `setDocumentLocator()`).
ContentHandler. **endDocument()**
Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler. **startPrefixMapping**(prefix, uri)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the `startPrefixMapping()` and `endPrefixMapping()` events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that `startPrefixMapping()` and `endPrefixMapping()` events are not guaranteed to be properly nested relative to each other: all `startPrefixMapping()` events will occur before the corresponding `startElement()` event, and all `endPrefixMapping()` events will occur after the corresponding `endElement()` event, but their order is not guaranteed.

ContentHandler. **endPrefixMapping**(prefix)
End the scope of a prefix-URI mapping.

See `startPrefixMapping()` for details. This event will always
occur after the corresponding `endElement()` event, but the order of `endPrefixMapping()` events is not otherwise guaranteed.

**ContentHandler.** `startElement(name, attrs)`  
Signals the start of an element in non-namespace mode.

The `name` parameter contains the raw XML 1.0 name of the element type as a string and the `attrs` parameter holds an object of the Attributes interface (see The Attributes Interface) containing the attributes of the element. The object passed as `attrs` may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the `attrs` object.

**ContentHandler.** `endElement(name)`  
Signals the end of an element in non-namespace mode.

The `name` parameter contains the name of the element type, just as with the `startElement()` event.

**ContentHandler.** `startElementNS(name, qname, attrs)`  
Signals the start of an element in namespace mode.

The `name` parameter contains the name of the element type as a `(uri, localname)` tuple, the `qname` parameter contains the raw XML 1.0 name used in the source document, and the `attrs` parameter parameter holds an instance of the AttributesNS interface (see The AttributesNS Interface) containing the attributes of the element. If no namespace is associated with the element, the `uri` component of `name` will be `None`. The object passed as `attrs` may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the
attributes, use the `copy()` method of the `attrs` object.

Parsers may set the `qname` parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

**ContentHandler.endElementNS(name, qname)**
Signals the end of an element in namespace mode.

The `name` parameter contains the name of the element type, just as with the `startElementNS()` method, likewise the `qname` parameter.

**ContentHandler.characters(content)**
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

`content` may be a string or bytes instance; the `expat` reader module always produces strings.

**Note:** The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use `content` instead of slicing content with the old `offset` and `length` parameters.

**ContentHandler.ignorableWhitespace(whitespace)**
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler. \texttt{processingInstruction(target, data)}
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler. \texttt{skippedEntity(name)}
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the \texttt{feature\_external\_ges} and the \texttt{feature\_external\_pes} properties.
DTDHandler Objects

**DTDHandler** instances provide the following methods:

`DTDHandler.notationDecl(name, publicId, systemId)`

Handle a notation declaration event.

`DTDHandler.unparsedEntityDecl(name, publicId, systemId, ndata)`

Handle an unparsed entity declaration event.
EntityResolver Objects

EntityResolver. **resolveEntity**(publicId, systemId)

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns `systemId`. 
ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the XMLReader. If you create an object that implements this interface, then register the object with your XMLReader, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a SAXParseException as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

**ErrorHandler.error(exception)**

Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

**ErrorHandler.fatalError(exception)**

Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

**ErrorHandler.warning(exception)**

Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.
The module `xml.sax.saxutils` contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

`xml.sax.saxutils.escape(data, entities={})`

Escape `&`, `<`, and `>` in a string of data.

You can escape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `&`, `<` and `>` are always escaped, even if `entities` is provided.

`xml.sax.saxutils.unescape(data, entities={})`

Unescape `&amp;`, `&lt;`, and `&gt;` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional `entities` parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `&amp;`, `&lt;`, and `&gt;` are always unescaped, even if `entities` is provided.

`xml.sax.saxutils.quoteattr(data, entities={})`

Similar to `escape()`, but also prepares `data` to be used as an attribute value. The return value is a quoted version of `data` with
any additional required replacements. \texttt{quoteattr()} will select a quote character based on the content of \texttt{data}, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in \texttt{data}, the double-quote characters will be encoded and \texttt{data} will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

\begin{verbatim}
>>> print("<element attr=%s>" % quoteattr("ab ' c
d<element attr="ab ' cd &quot; ef")

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

class xml.sax.saxutils.XMLGenerator(out=None,
encoding='iso-8859-1', short_empty_elements=False)

This class implements the \texttt{ContentHandler} interface by writing SAX events back into an XML document. In other words, using an \texttt{XMLGenerator} as the content handler will reproduce the original document being parsed. \texttt{out} should be a file-like object which will default to \texttt{sys.stdout}. \texttt{encoding} is the encoding of the output stream which defaults to \texttt{'iso-8859-1'}. \texttt{short_empty_elements} controls the formatting of elements that contain no content: if \texttt{False} (the default) they are emitted as a pair of start/end tags, if set to \texttt{True} they are emitted as a single self-closed tag.

\textit{New in version 3.2: The short\_empty\_elements parameter.}

class xml.sax.saxutils.XMLFilterBase(base)

This class is designed to sit between an \texttt{XMLReader} and the client application’s event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers
unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

```python
xml.sax.saxutils.prepare_input_source(source, base='')
```

This function takes an input source and an optional base URL and returns a fully resolved `InputSource` object ready for reading. The input source can be given as a string, a file-like object, or an `InputSource` object; parsers will use this function to implement the polymorphic `source` argument to their `parse()` method.
XML SAX XML Reader — Interface for XML parsers

Source code: Lib/xml/sax/xmlreader.py

SAX parsers implement the XMLReader interface. They are implemented in a Python module, which must provide a function create_parser(). This function is invoked by xml.sax.make_parser() with no arguments to create a new parser object.

class xml.sax.xmlreader.XMLReader
    Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.IncrementalParser
    In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still parse() won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of parse() is not desirable.

    When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

    Note that these methods must not be called during parsing, that is, after parse has been called and before it returns.
By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

*class* `xml.sax.xmlreader.Locator`

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return None.

*class* `xml.sax.xmlreader.InputSource(system_id=None)`

Encapsulation of the information needed by the `XMLReader` to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the `XMLReader.parse()` method and for returning from `EntityResolver.resolveEntity`.

An `InputSource` belongs to the application, the `XMLReader` is not allowed to modify `InputSource` objects passed to it from the application, although it may make copies and modify those.

*class* `xml.sax.xmlreader.AttributesImpl(attrs)`

This is an implementation of the Attributes interface (see section `The Attributes Interface`). This is a dictionary-like object which represents the element attributes in a `startElement` call. In addition to the most useful dictionary operations, it supports
a number of other methods as described by the interface. Objects of this class should be instantiated by readers; `attrs` must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.AttributesNSImpl(attrs, qnames)
Namespace-aware variant of `AttributesImpl`, which will be passed to `startElementNS()`. It is derived from `AttributesImpl`, but understands attribute names as two-tuples of `namespaceURI` and `localname`. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the AttributesNS interface (see section The AttributesNS Interface).
XMLReader Objects

The XMLReader interface supports the following methods:

XMLReader.parse(source)
Process an input source, producing SAX events. The source object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an InputSource object. When parse() returns, the input is completely processed, and the parser object can be discarded or reset.

Changed in version 3.5: Added support of character streams.

XMLReader.getContentHandler()
Return the current ContentHandler.

XMLReader.setContentHandler(handler)
Set the current ContentHandler. If no ContentHandler is set, content events will be discarded.

XMLReader.getDTDHandler()
Return the current DTDHandler.

XMLReader.setDTDHandler(handler)
Set the current DTDHandler. If no DTDHandler is set, DTD events will be discarded.

XMLReader.getEntityResolver()
Return the current EntityResolver.

XMLReader.setEntityResolver(handler)
Set the current `EntityResolver`. If no `EntityResolver` is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader. **getErrorHandler()**
Return the current `ErrorHandler`.

XMLReader. **setErrorHandler(handler)**
Set the current error handler. If no `ErrorHandler` is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader. **setLocale(locale)**
Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader. **getFeature(featurename)**
Return the current setting for feature `featurename`. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module `xml.sax.handler`.

XMLReader. **setFeature(featurename, value)**
Set the `featurename` to `value`. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, `SAXNotSupportedException` is raised.

XMLReader. **getProperty(propertyname)**
Return the current setting for property `propertyname`. If the
property is not recognized, a SAXNotRecognizedException is raised. The well-known property names are listed in the module `xml.sax.handler`.

**XMLReader.set**Property**(propertyname, value)**

Set the property name to value. If the property is not recognized, SAXNotRecognizedException is raised. If the property or its setting is not supported by the parser, SAXNotSupported**Exception** is raised.
IncrementalParser Objects

Instances of `IncrementalParser` offer the following additional methods:

IncrementalParser.**feed**(data)

Process a chunk of `data`.

IncrementalParser.**close**()

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.**reset**()

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.
Locator Objects

Instances of `Locator` provide these methods:

- `Locator.getColumnNumber()`  
  Return the column number where the current event begins.

- `Locator.getLineNumber()`    
  Return the line number where the current event begins.

- `Locator.getPublicId()`      
  Return the public identifier for the current event.

- `Locator.getSystemId()`      
  Return the system identifier for the current event.
InputSource Objects

InputSource.setPublicId(id)
    Sets the public identifier of this InputSource.

InputSource.getPublicId()
    Returns the public identifier of this InputSource.

InputSource.setSystemId(id)
    Sets the system identifier of this InputSource.

InputSource.getSystemId()
    Returns the system identifier of this InputSource.

InputSource.setEncoding(encoding)
    Sets the character encoding of this InputSource.
    The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).
    The encoding attribute of the InputSource is ignored if the InputSource also contains a character stream.

InputSource.getEncoding()
    Get the character encoding of this InputSource.

InputSource.setByteStream(bytefile)
    Set the byte stream (a binary file) for this input source.
    The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.
If the application knows the character encoding of the byte stream, it should set it with the `setEncoding` method.

**InputSource.** `getByteStream()`  
Get the byte stream for this input source.

The `getEncoding` method will return the character encoding for this byte stream, or `None` if unknown.

**InputSource.** `setCharacterStream(charfile)`  
Set the character stream (a text file) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

**InputSource.** `getCharacterStream()`  
Get the character stream for this input source.
The Attributes Interface

Attributes objects implement a portion of the mapping protocol, including the methods copy(), get(), __contains__(), items(), keys(), and values(). The following methods are also provided:

Attributes.**getLength**()
   Return the number of attributes.

Attributes.**getNames**()
   Return the names of the attributes.

Attributes.**getType**(name)
   Returns the type of the attribute name, which is normally 'CDATA'.

Attributes.**getValue**(name)
   Return the value of attribute name.
The AttributesNS Interface

This interface is a subtype of the Attributes interface (see section The Attributes Interface). All methods supported by that interface are also available on AttributesNS objects.

The following methods are also available:

AttributesNS.getValueByQName(name)
Return the value for a qualified name.

AttributesNS.getNameByQName(name)
Return the (namespace, localname) pair for a qualified name.

AttributesNS.getQNameByName(name)
Return the qualified name for a (namespace, localname) pair.

AttributesNS.getQNames()
Return the qualified names of all attributes.
The xml.parsers.expat module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

(exception) xml.parsers.expat.ExpatError

The exception raised when Expat reports an error. See section ExpatError Exceptions for more information on interpreting Expat errors.

(exception) xml.parsers.expat.error

Alias for ExpatError.

xml.parsers.expat.XMLParserType

The type of the return values from the ParserCreate() function.
The `xml.parsers.expat` module contains two functions:

```python
xml.parsers.expat.ErrorString(errno)
```
Returns an explanatory string for a given error number `errno`.

```python
xml.parsers.expat.ParserCreate(encoding=None, namespace_separator=None)
```
Creates and returns a new `xmlparser` object. `encoding`, if specified, must be a string naming the encoding used by the XML data. Expat doesn’t support as many encodings as Python does, and its repertoire of encodings can’t be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If `encoding` [1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for `namespace_separator`. The value must be a one-character string; a `ValueError` will be raised if the string has an illegal length (`None` is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers `StartElementHandler` and `EndElementHandler` will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (`chr(0)`) then the namespace URI and the local part will be concatenated without any separator.

For example, if `namespace_separator` is set to a space character (`' '`) and the following document is parsed:

```xml
<?xml version="1.0"?>
<root xmlns="http://default-namespace.org/
xmlns:py = "http://www.python.org/ns/">
```
StartElementHandler will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the Expat library used by pyexpat, the xmlparser instance returned can only be used to parse a single XML document. Call ParserCreate for each document to provide unique parser instances.

**See also:**

**The Expat XML Parser**

Home page of the Expat project.
XMLParser Objects

xmlparser objects have the following methods:

xmlparser.**Parse**(data[, isfinal])

Parses the contents of the string data, calling the appropriate handler functions to process the parsed data. isfinal must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. data can be the empty string at any time.

xmlparser.**ParseFile**(file)

Parse XML data reading from the object file. file only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

xmlparser.**SetBase**(base)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the base argument to the ExternalEntityRefHandler(), NotationDeclHandler(), and UnparsedEntityDeclHandler() functions.

xmlparser.**GetBase**()

Returns a string containing the base set by a previous call to SetBase(), or None if SetBase() hasn't been called.

xmlparser.**GetInputContext**()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text.
When called while an event handler is not active, the return value is None.

`xmlparser.ExternalEntityParserCreate(context[, encoding])`
Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The `context` parameter should be the string passed to the `ExternalEntityRefHandler()` handler function, described below. The child parser is created with the `ordered_attributes` and `specified_attributes` set to the values of this parser.

`xmlparser.SetParamEntityParsing(flag)`
Control parsing of parameter entities (including the external DTD subset). Possible `flag` values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

`xmlparser.UseForeignDTD([flag])`
Calling this with a true value for `flag` (the default) will cause Expat to call the `ExternalEntityRefHandler` with `None` for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the `ExternalEntityRefHandler` will still be called, but the `StartDoctypeDeclHandler` and `EndDoctypeDeclHandler` will not be called.

Passing a false value for `flag` will cancel a previous call that passed a true value, but otherwise has no effect.
This method can only be called before the `Parse()` or `ParseFile()` methods are called; calling it after either of those have been called causes `ExpatError` to be raised with the code set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE]`.

```
xmlobjects have the following attributes:

```
xmlobject.buffer_size
```
The size of the buffer used when `buffer_text` is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

```
xmlobject.buffer_text
```
Setting this to true causes the `xmlobject` object to buffer textual content returned by Expat to avoid multiple calls to the `CharacterDataHandler()` callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

```
xmlobject.buffer_used
```
If `buffer_text` is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when `buffer_text` is false.

```
xmlobject.ordered_attributes
```
Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this
format.) By default, this attribute is false; it may be changed at any time.

`xmlparser.specified_attributes`
If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an `xml.parsers.expat.ExpatError` exception.

`xmlparser.ErrorByteIndex`
Byte index at which an error occurred.

`xmlparser.ErrorCode`
Numeric code specifying the problem. This value can be passed to the `ErrorString()` function, or compared to one of the constants defined in the `errors` object.

`xmlparser.ErrorColumnNumber`
Column number at which an error occurred.

`xmlparser.ErrorLineNumber`
Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse
event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

`xmlparser.CurrentByteIndex`
Current byte index in the parser input.

`xmlparser.CurrentColumnNumber`
Current column number in the parser input.

`xmlparser.CurrentLineNumber`
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object `o`, use `o.handlername = func`. `handlername` must be taken from the following list, and `func` must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

`xmlparser.XmlDeclHandler(version, encoding, standalone)`
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional “standalone” declaration. `version` and `encoding` will be strings, and `standalone` will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

`xmlparser.StartDoctypeDeclHandler(doctypeName, systemId, publicId, has_internal_subset)`
Called when Expat begins parsing the document type declaration
(<!DOCTYPE ...). The doctypeName is provided exactly as presented. The systemId and publicId parameters give the system and public identifiers if specified, or None if omitted. has_internal_subset will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

`xmlparser.EndDoctypeDeclHandler()`

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

`xmlparser.ElementDeclHandler(name, model)`

Called once for each element type declaration. name is the name of the element type, and model is a representation of the content model.

`xmlparser.AttlistDeclHandler(elname, attname, type, default, required)`

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. elname is the name of the element to which the declaration applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the possible values are 'CDATA', 'ID', 'IDREF', ... default gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, required will be true. This requires Expat version 1.95.0 or newer.

`xmlparser.StartElementHandler(name, attributes)`

Called for the start of every element. name is a string containing
the element name, and attributes is the element attributes. If ordered_attributes is true, this is a list (see ordered_attributes for a full description). Otherwise it's a dictionary mapping names to values.

```
xmlparser.EndElementHandler(name)

Called for the end of every element.
```

```
xmlparser.ProcessingInstructionHandler(target, data)

Called for every processing instruction.
```

```
xmlparser.CharacterDataHandler(data)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler callbacks to collect the required information.
```

```
xmlparser.UnparsedEntityDeclHandler(entityName, base, systemId, publicId, notationName)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use EntityDeclHandler instead. (The underlying function in the Expat library has been declared obsolete.)
```

```
xmlparser.EntityDeclHandler(entityName, is_parameter_entity, value, base, systemId, publicId, notationName)

Called for all entity declarations. For parameter and internal entities, value will be a string giving the declared contents of the entity; this will be None for external entities. The notationName
parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. `is_parameter_entity` will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

```python
xmlparser. NotationDeclHandler(notationName, base, systemId, publicId)
```
Called for notation declarations. `notationName`, `base`, and `systemId`, and `publicId` are strings if given. If the public identifier is omitted, `publicId` will be `None`.

```python
xmlparser. StartNamespaceDeclHandler(prefix, uri)
```
Called when an element contains a namespace declaration. Namespace declarations are processed before the `StartElementHandler` is called for the element on which declarations are placed.

```python
xmlparser. EndNamespaceDeclHandler(prefix)
```
Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the `StartNamespaceDeclHandler` was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding `EndElementHandler` for the end of the element.

```python
xmlparser. CommentHandler(data)
```
Called for comments. `data` is the text of the comment, excluding the leading `'<!'` and trailing `'>-'`.

```python
xmlparser. StartCdataSectionHandler()
```
Called at the start of a CDATA section. This and `EndCdataSectionHandler` are needed to be able to identify the syntactical start and end for CDATA sections.

```xml
xmlparser.EndCdataSectionHandler()
```
Called at the end of a CDATA section.

```xml
xmlparser.DefaultHandler(data)
```
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

```xml
xmlparser.DefaultHandlerExpand(data)
```
This is the same as the `DefaultHandler()`, but doesn’t inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

```xml
xmlparser.NotStandaloneHandler()
```
Called if the XML document hasn’t been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

```xml
xmlparser.ExternalEntityRefHandler(context, base, systemId, publicId)
```
Called for references to external entities. `base` is the current base, as set by a previous call to `SetBase()`. The public and system identifiers, `systemId` and `publicId`, are strings if given; if the public
identifier is not given, publicId will be None. The context value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using ExternalEntityParserCreate(context), initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will raise an XML_ERROR_EXTERNAL_ENTITY_HANDLING error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the DefaultHandler callback, if provided.
ExpatError Exceptions

**ExpatError** exceptions have a number of interesting attributes:

**ExpatError.code**

Expat's internal error number for the specific error. The **errors.messages** dictionary maps these error numbers to Expat's error messages. For example:

```python
from xml.parsers.expat import ParserCreate, ExpatError
p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The **errors** module also provides error message constants and a dictionary **codes** mapping these messages back to the error codes, see below.

**ExpatError.lineno**

Line number on which the error was detected. The first line is numbered 1.

**ExpatError.offset**

Character offset into the line where the error occurred. The first column is numbered 0.
Example

The following program defines three handlers that just print out their arguments.

```python
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()
p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version='1.0'?><parent id="top"><child1 name="paul">Text goes here
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '
'
Start element: child2 {'name': 'fred'}
```
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the `xml.parsers.expat.model` module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

- `xml.parsers.expat.model.XML_CTYPE_ANY`
  The element named by the model name was declared to have a content model of `ANY`.

- `xml.parsers.expat.model.XML_CTYPE_CHOICE`
  The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

- `xml.parsers.expat.model.XML_CTYPE_EMPTY`
  Elements which are declared to be `EMPTY` have this model type.

- `xml.parsers.expat.model.XML_CTYPE_MIXED`

- `xml.parsers.expat.model.XML_CTYPE_NAME`

- `xml.parsers.expat.model.XML_CTYPE_SEQ`
  Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:
xml.parsers.expat.model.XML_CQUANT_NONE
No modifier is given, so it can appear exactly once, as for A.

xml.parsers.expat.model.XML_CQUANT_OPT
The model is optional: it can appear once or not at all, as for A?.

xml.parsers.expat.model.XML_CQUANT_PLUS
The model must occur one or more times (like A+).

xml.parsers.expat.model.XML_CQUANT_REP
The model must occur zero or more times, as for A*.
Expat error constants

The following constants are provided in the xml.parsers.expat.errors module. These constants are useful in interpreting some of the attributes of the ExpatError exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error message and not the numeric error code, you do this by comparing its code attribute with errors.codes[errors.XML_ERROR_CONSTANT_NAME].

The errors module has the following attributes:

xml.parsers.expat.errors.codes
A dictionary mapping numeric error codes to their string descriptions.

    New in version 3.2.

xml.parsers.expat.errors.messages
A dictionary mapping string descriptions to their error codes.

    New in version 3.2.

xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY
xml.parsers.expat.errors.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.XML_ERROR_BAD_CHAR_REF
A character reference referred to a character which is illegal in XML
(for example, character ₀, or 'Debees').

```python
xml.parsers.expat.errors.XML_ERROR_BINARY_ENTITY_REF
An entity reference referred to an entity which was declared with a notation, so cannot be parsed.
```

```python
xml.parsers.expat.errors.XML_ERROR_DUPLICATE_ATTRIBUTE
An attribute was used more than once in a start tag.
```

```python
xml.parsers.expat.errors.XML_ERROR_INCORRECT_ENCODING
xml.parsers.expat.errors.XML_ERROR_INVALID_TOKEN
Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value ₀) in a UTF-8 input stream.
```

```python
xml.parsers.expat.errors.XML_ERROR_JUNK_AFTER_DOC_ELEMENT
Something other than whitespace occurred after the document element.
```

```python
xml.parsers.expat.errors.XML_ERROR_MISPLACED_XML_PI
An XML declaration was found somewhere other than the start of the input data.
```

```python
xml.parsers.expat.errors.XML_ERROR_NO_ELEMENTS
The document contains no elements (XML requires all documents to contain exactly one top-level element).
```

```python
xml.parsers.expat.errors.XML_ERROR_NO_MEMORY
Expat was not able to allocate memory internally.
```

```python
xml.parsers.expat.errors.XML_ERROR_PARAM_ENTITY_REF
A parameter entity reference was found where it was not allowed.
```
An incomplete character was found in the input.

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

Some unspecified syntax error was encountered.

An end tag did not match the innermost open start tag.

Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

A reference was made to an entity which was not defined.

The document encoding is not supported by Expat.

A CDATA marked section was not closed.

The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the NotStandaloneHandler was set and returned 0.

The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the NotStandaloneHandler was set and returned 0.
xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE

An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the xml.parsers.expat module.

xml.parsers.expat.errors.XML_ERROR_FEATURE_REQUIRES_XML_DTD

A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by UseForeignDTD().

xml.parsers.expat.errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.XML_ERROR_TEXT_DECL

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.XML_ERROR_PUBLICID

Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.XML_ERROR_XML_DECL

The document contained no document element at all.

xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX

The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.XML_ERROR_UNBOUND_PREFIX

An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.XML_ERROR_SUSPENDED

Characters were found in the public id that are not allowed.
The requested operation was made on a suspended parser, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.

```python
xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED
```

An attempt to resume the parser was made when the parser had not been suspended.

```python
xml.parsers.expat.errors.XML_ERROR_ABORTED
```

This should not be reported to Python applications.

```python
xml.parsers.expat.errors.XML_ERROR_FINISHED
```

The requested operation was made on a parser which was finished parsing input, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.

```python
xml.parsers.expat.errors.XML_ERROR_SUSPEND_PE
```

Footnotes

[1] The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See [https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl](https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl) and [https://www.iana.org/assignments/character-sets/character-sets.xhtml](https://www.iana.org/assignments/character-sets/character-sets.xhtml).
Internet Protocols and Support

The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module `socket`, which is currently supported on most popular platforms. Here is an overview:

- **webbrowser** — Convenient Web-browser controller
  - Browser Controller Objects
- **cgi** — Common Gateway Interface support
  - Introduction
  - Using the cgi module
  - Higher Level Interface
  - Functions
  - Caring about security
  - Installing your CGI script on a Unix system
  - Testing your CGI script
  - Debugging CGI scripts
  - Common problems and solutions
- **cgitb** — Traceback manager for CGI scripts
- **wsgiref** — WSGI Utilities and Reference Implementation
  - `wsgiref.util` — WSGI environment utilities
  - `wsgiref.headers` — WSGI response header tools
  - `wsgiref.simple_server` — a simple WSGI HTTP server
  - `wsgiref.validate` — WSGI conformance checker
  - `wsgiref.handlers` — server/gateway base classes
  - Examples
- **urllib** — URL handling modules
- **urllib.request** — Extensible library for opening URLs
  - Request Objects
  - OpenerDirector Objects
- BaseHandler Objects
- HTTPRedirectHandler Objects
- HTTPCookieProcessor Objects
- ProxyHandler Objects
- HTTPPasswordMgr Objects
- HTTPPasswordMgrWithPriorAuth Objects
- AbstractBasicAuthHandler Objects
- HTTPBasicAuthHandler Objects
- ProxyBasicAuthHandler Objects
- AbstractDigestAuthHandler Objects
- HTTPDigestAuthHandler Objects
- ProxyDigestAuthHandler Objects
- HTTPHandler Objects
- HTTPSHandler Objects
- FileHandler Objects
- DataHandler Objects
- FTPHandler Objects
- CacheFTPHandler Objects
- UnknownHandler Objects
- HTTPErrorProcessor Objects
- Examples
- Legacy interface
- urllib.request Restrictions

- urllib.response — Response classes used by urllib
- urllib.parse — Parse URLs into components
  - URL Parsing
  - Parsing ASCII Encoded Bytes
  - Structured Parse Results
  - URL Quoting
- urllib.error — Exception classes raised by urllib.request
- urllib.robotparser — Parser for robots.txt
- http — HTTP modules
  - HTTP status codes
- http.client — HTTP protocol client
- HTTPConnection Objects
- HTTPResponse Objects
- Examples
- HTTPMessage Objects
- ftplib — FTP protocol client
  - FTP Objects
  - FTP_TLS Objects
- poplib — POP3 protocol client
  - POP3 Objects
  - POP3 Example
- imaplib — IMAP4 protocol client
  - IMAP4 Objects
  - IMAP4 Example
- nntplib — NNTP protocol client
  - NNTP Objects
    - Attributes
    - Methods
  - Utility functions
- smtplib — SMTP protocol client
  - SMTP Objects
  - SMTP Example
- smtpd — SMTP Server
  - SMTPServer Objects
  - DebuggingServer Objects
  - PureProxy Objects
  - MailmanProxy Objects
  - SMTPChannel Objects
- telnetlib — Telnet client
  - Telnet Objects
  - Telnet Example
- uuid — UUID objects according to RFC 4122
  - Example
- socketserver — A framework for network servers
- Server Creation Notes
- Server Objects
- Request Handler Objects
- Examples
  - `socketserver.TCPServer` Example
  - `socketserver.UDPServer` Example
  - Asynchronous Mixins
- `http.server` — HTTP servers
- `http.cookies` — HTTP state management
  - Cookie Objects
  - Morsel Objects
  - Example
- `http.cookiejar` — Cookie handling for HTTP clients
  - CookieJar and FileCookieJar Objects
  - FileCookieJar subclasses and co-operation with web browsers
  - CookiePolicy Objects
  - DefaultCookiePolicy Objects
  - Cookie Objects
  - Examples
- `xmlrpc` — XMLRPC server and client modules
- `xmlrpc.client` — XML-RPC client access
  - ServerProxy Objects
  - DateTime Objects
  - Binary Objects
  - Fault Objects
  - ProtocolError Objects
  - MultiCall Objects
  - Convenience Functions
  - Example of Client Usage
  - Example of Client and Server Usage
- `xmlrpc.server` — Basic XML-RPC servers
  - SimpleXMLRPCServer Objects
- CGIXMLRPCRequestHandler
- Documenting XMLRPC server
- DocXMLRPCServer Objects
- DocCGIXMLRPCRequestHandler
- `ipaddress` — IPv4/IPv6 manipulation library
  - Convenience factory functions
  - IP Addresses
    - Address objects
    - Conversion to Strings and Integers
    - Operators
      - Comparison operators
      - Arithmetic operators
  - IP Network definitions
    - Prefix, net mask and host mask
    - Network objects
    - Operators
      - Logical operators
      - Iteration
      - Networks as containers of addresses
  - Interface objects
    - Operators
      - Logical operators
  - Other Module Level Functions
  - Custom Exceptions
webbrowser — Convenient Web-browser controller

Source code: Lib/webbrowser.py

The `webbrowser` module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn’t available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted as the `os.pathsep`-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch. [1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script `webbrowser` can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following
optional parameters: -n opens the URL in a new browser window, if possible; -t opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "http://www.python.org"
```

The following exception is defined:

```
exception webbrowser.Error
    Exception raised when a browser control error occurs.
```

The following functions are defined:

```
webbrowser.open(url, new=0, autoraise=True)
    Display url using the default browser. If new is 0, the url is opened
    in the same browser window if possible. If new is 1, a new browser
    window is opened if possible. If new is 2, a new browser page
    ("tab") is opened if possible. If autoraise is True, the window is
    raised if possible (note that under many window managers this will
    occur regardless of the setting of this variable).

    Note that on some platforms, trying to open a filename using this
    function, may work and start the operating system's associated
    program. However, this is neither supported nor portable.

webbrowser.open_new(url)
    Open url in a new window of the default browser, if possible,
    otherwise, open url in the only browser window.

webbrowser.open_new_tab(url)
    Open url in a new page ("tab") of the default browser, if possible,
    otherwise equivalent to open_new().```
webbrowser.get(using=None)

Return a controller object for the browser type using. If using is None, return a controller for a default browser appropriate to the caller’s environment.

webbrowser.register(name, constructor, instance=None, *, preferred=False)

Register the browser type name. Once a browser type is registered, the get() function can return a controller for that browser type. If instance is not provided, or is None, constructor will be called without parameters to create an instance when needed. If instance is provided, constructor will never be called, and may be None.

Setting preferred to True makes this browser a preferred result for a get() call with no argument. Otherwise, this entry point is only useful if you plan to either set the BROWSER variable or call get() with a nonempty argument matching the name of a handler you declare.

Changed in version 3.7: preferred keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the get() function and the corresponding instantiations for the controller classes, all defined in this module.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Class Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'mozilla'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'firefox'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>Browser Name</td>
<td>Classname</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>'netscape'</td>
<td>Mozilla('netscape')</td>
<td></td>
</tr>
<tr>
<td>'galeon'</td>
<td>Galeon('galeon')</td>
<td></td>
</tr>
<tr>
<td>'epiphany'</td>
<td>Galeon('epiphany')</td>
<td></td>
</tr>
<tr>
<td>'skipstone'</td>
<td>BackgroundBrowser('skipstone')</td>
<td></td>
</tr>
<tr>
<td>'kfmclient'</td>
<td>Konqueror()</td>
<td></td>
</tr>
<tr>
<td>'konqueror'</td>
<td>Konqueror()</td>
<td></td>
</tr>
<tr>
<td>'kfm'</td>
<td>Konqueror()</td>
<td></td>
</tr>
<tr>
<td>'mosaic'</td>
<td>BackgroundBrowser('mosaic')</td>
<td></td>
</tr>
<tr>
<td>'opera'</td>
<td>Opera()</td>
<td></td>
</tr>
<tr>
<td>'grail'</td>
<td>Grail()</td>
<td></td>
</tr>
<tr>
<td>'links'</td>
<td>GenericBrowser('links')</td>
<td></td>
</tr>
<tr>
<td>'elinks'</td>
<td>Elinks('elinks')</td>
<td></td>
</tr>
<tr>
<td>'lynx'</td>
<td>GenericBrowser('lynx')</td>
<td></td>
</tr>
<tr>
<td>'w3m'</td>
<td>GenericBrowser('w3m')</td>
<td></td>
</tr>
<tr>
<td>'windows-default'</td>
<td>WindowsDefault</td>
<td></td>
</tr>
<tr>
<td>'macosx'</td>
<td>MacOSX('default')</td>
<td></td>
</tr>
<tr>
<td>'safari'</td>
<td>MacOSX('safari')</td>
<td></td>
</tr>
<tr>
<td>'google-chrome'</td>
<td>Chrome('google-chrome')</td>
<td></td>
</tr>
<tr>
<td>'chrome'</td>
<td>Chrome('chrome')</td>
<td></td>
</tr>
<tr>
<td>'chromium'</td>
<td>Chromium('chromium')</td>
<td></td>
</tr>
<tr>
<td>'chromium-browser'</td>
<td>Chromium('chromium-browser')</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. “Konqueror” is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the KDEDIR variable is not sufficient. Note also that the name “kfm” is used even when using the konqueror command with KDE 2 — the implementation selects the best strategy for running Konqueror.

2. Only on Windows platforms.
3. Only on Mac OS X platform.

New in version 3.3: Support for Chrome/Chromium has been added.

Here are some simple examples:

```python
url = 'http://docs.python.org/

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```
Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

controller. **open**(url, new=0, autoraise=True)

Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page (“tab”) is opened if possible.

controller. **open_new**(url)

Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias **open_new()**.

controller. **open_new_tab**(url)

Open *url* in a new page (“tab”) of the browser handled by this controller, if possible, otherwise equivalent to **open_new()**.

**Footnotes**

[1] Executables named here without a full path will be searched in the directories given in the PATH environment variable.
cgi — Common Gateway Interface support

Source code: Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.
Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server’s special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client’s hostname, the requested URL, the query string, and lots of other goodies) in the script’s shell environment, executes the script, and sends the script’s output back to the client.

The script's input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the “query string” part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```python
print("Content-Type: text/html")  # HTML is followed
print()  # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here's Python code that prints a simple piece of HTML:

```python
print("<TITLE>CGI script output</TITLE>")
```
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
Using the CGI module

Begin by writing `import cgi`.

When you write a new script, consider adding these lines:

```python
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```python
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by `cgitb` provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, use the `FieldStorage` class. If the form contains non-ASCII characters, use the `encoding` keyword parameter set to the value of the encoding defined for the document. It is usually contained in the META tag in the HEAD section of the HTML document or by the `Content-Type` header). This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.
The FieldStorage instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method `keys()` and the built-in function `len()`. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional `keep_blank_values` keyword parameter when creating the FieldStorage instance.

For instance, the following code (which assumes that the `Content-Type` header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```python
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>"
    print("Please fill in the name and addr fields.
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of FieldStorage (or MiniFieldStorage, depending on the form encoding). The value attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a FieldStorage or MiniFieldStorage instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If
you expect this possibility (when your HTML form contains multiple fields with the same name), use the `getlist()` method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```python
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the value attribute or the `getvalue()` method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the filename attribute or the `file` attribute. You can then read the data from the `file` attribute before it is automatically closed as part of the garbage collection of the `FieldStorage` instance (the `read()` and `readline()` methods will return bytes):

```python
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
    linecount = linecount + 1
```

FieldStorage objects also support being used in a `with` statement, which will automatically close them when done.

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object
for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive multipart/* encoding). When this occurs, the item will be a dictionary-like FieldStorage item. This can be determined by testing its type attribute, which should be multipart/form-data (or perhaps another MIME type matching multipart/*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the “old” format (as the query string or as a single data part of type application/x-www-form-urlencoded), the items will actually be instances of the class MiniFieldStorage. In this case, the list, file, and filename attributes are always None.

A form submitted via POST that also has a query string will contain both FieldStorage and MiniFieldStorage items.

*Changed in version 3.4:* The file attribute is automatically closed upon the garbage collection of the creating FieldStorage instance.

*Changed in version 3.5:* Added support for the context management protocol to the FieldStorage class.
Higher Level Interface

The previous section explains how to read CGI form data using the FieldStorage class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn’t make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```python
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```html
<input type="checkbox" name="item" value="1" />  
<input type="checkbox" name="item" value="2" />  
```

In most situations, however, there’s only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:
user = form.getvalue("user").upper()

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another user=foo pair to the query string, then the script would crash, because in this situation the getvalue("user") method call returns a list instead of a string. Calling the upper() method on a list is not valid (since lists do not have a method of this name) and results in an AttributeError exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods getfirst() and getlist() provided by this higher level interface.

FieldStorage.getfirst(name, default=None)

This method always returns only one value associated with form field name. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on. [1] If no such form field or value exists then the method returns the value specified by the optional parameter default. This parameter defaults to None if not specified.

FieldStorage.getlist(name)

This method always returns a list of values associated with form field name. The method returns an empty list if no such form field or value exists for name. It returns a list consisting of one item if
only one such value exists.

Using these methods you can write nice compact code:

```python
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()  # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```
Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

\[
\text{cgi.} \text{parse}(\text{fp=\text{None}, environ=\text{os.environ}, keep\_blank\_values=\text{False}, strict\_parsing=\text{False})}
\]

Parse a query in the environment or from a file (the file defaults to \text{sys.stdin}). The \text{keep\_blank\_values} and \text{strict\_parsing} parameters are passed to \text{urllib.parse.parse\_qs()} unchanged.

\[
\text{cgi.} \text{parse\_qs}(\text{qs, keep\_blank\_values=\text{False}, strict\_parsing=\text{False})}
\]

This function is deprecated in this module. Use \text{urllib.parse.parse\_qs()} instead. It is maintained here only for backward compatibility.

\[
\text{cgi.} \text{parse\_qs1}(\text{qs, keep\_blank\_values=\text{False}, strict\_parsing=\text{False})}
\]

This function is deprecated in this module. Use \text{urllib.parse.parse\_qs1()} instead. It is maintained here only for backward compatibility.

\[
\text{cgi.} \text{parse\_multipart}(\text{fp, pdict, encoding=\text{\textquote{utf-8}}, errors=\text{\textquote{replace}}})
\]

Parse input of type \text{multipart/form-data} (for file uploads). Arguments are \text{fp} for the input file, \text{pdict} for a dictionary containing other parameters in the \text{Content-Type} header, and \text{encoding}, the request encoding.
Returns a dictionary just like `urllib.parse.parse_qs()`: keys are the field names, each value is a list of values for that field. For non-file fields, the value is a list of strings.

This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the FieldStorage class instead which is much more flexible.

*Changed in version 3.7:* Added the `encoding` and `errors` parameters. For non-file fields, the value is now a list of strings, not bytes.

```python
cgi.parse_header(string)
```
Parse a MIME header (such as `Content-Type`) into a main value and a dictionary of parameters.

```python
cgi.test()
```
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

```python
cgi.print_environ()
```
Format the shell environment in HTML.

```python
cgi.print_form(form)
```
Format a form in HTML.

```python
cgi.print_directory()
```
Format the current directory in HTML.

```python
cgi.print_environ_usage()
```
Print a list of useful (used by CGI) environment variables in HTML.
cgi.escape(s, quote=False)

Convert the characters '&', '<' and '>' in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag quote is true, the quotation mark character (") is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in <a href="...">. Note that single quotes are never translated.

Deprecated since version 3.2: This function is unsafe because quote is false by default, and therefore deprecated. Use html.escape() instead.
Caring about security

There’s one important rule: if you invoke an external program (via the \texttt{os.system()} or \texttt{os.popen()} functions. or others with similar functionality), make very sure you don’t pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn’t have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.
Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by “others”; the Unix file mode should be `0o755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by “others”.

Make sure that any files your script needs to read or write are readable or writable, respectively, by “others” — their mode should be `0o644` for readable and `0o666` for writable. This is because, for security reasons, the HTTP server executes your script as user “nobody”, without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server’s `cgi-bin` directory) and the set of environment variables is also different from what you get when you log in. In particular, don’t count on the shell’s search path for executables (PATH) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python’s default module search path, you can change the path in your script,
before importing other modules. For example:

```python
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server’s documentation (it will usually have a section on CGI scripts).
Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There’s one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won’t execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.
Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (cgi.py) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it’s installed in the standard cgi-bin directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there’s an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as “addr” with value “At Home” and “name” with value “Joe Blow”), the cgi.py script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the cgi module’s test() function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the cgi.py file itself.
When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, you can easily send tracebacks to the Web browser using the cgitb module. If you haven’t done so already, just add the lines:

```python
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the cgitb module, you can use an even more robust approach (which only uses built-in modules):

```python
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML
interpretation is going on, the traceback will be readable.
Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client’s display while the script is running.
- Check the installation instructions above.
- Check the HTTP server’s log files. *(tail -f logfile in a separate window may be useful!)*
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — PATH is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server’s suexec feature.
- Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

Footnotes

[1] Note that some recent versions of the HTML specification do not state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.
The `cgitb` module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add this to the top of your CGI script:

```python
import cgitb

cgitb.enable()
```

The options to the `enable()` function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

```python

cgitb.enable(display=1, logdir=None, context=5, format="html")
```

This function causes the `cgitb` module to take over the interpreter’s default handling for exceptions by setting the value of `sys.excepthook`. 
The optional argument *display* defaults to 1 and can be set to 0 to suppress sending the traceback to the browser. If the argument *logdir* is present, the traceback reports are written to files. The value of *logdir* should be a directory where these files will be placed. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5. If the optional argument *format* is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

**cgitb.text(info, context=5)**

This function handles the exception described by *info* (a 3-tuple containing the result of *sys.exc_info()*) formatting its traceback as text and returning the result as a string. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5.

**cgitb.html(info, context=5)**

This function handles the exception described by *info* (a 3-tuple containing the result of *sys.exc_info()*) formatting its traceback as HTML and returning the result as a string. The optional argument *context* is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5.

**cgitb.handler(info=None)**

This function handles an exception using the default settings (that is, show a report in the browser, but don't log to a file). This can be used when you've caught an exception and want to report it using cgitb. The optional *info* argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly
like the tuple returned by `sys.exc_info()`. If the `info` argument is not supplied, the current exception is obtained from `sys.exc_info()`.
The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don’t need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 3333).

See wsgi.readthedocs.io for more information about WSGI, and links to tutorials and other resources.
**wsgiref.util** – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 3333. All of the functions taking an `environ` parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 3333 for a detailed specification.

**wsgiref.util.guess_scheme(environ)**

Return a guess for whether `wsgi.url_scheme` should be “http” or “https”, by checking for a HTTPS environment variable in the `environ` dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of “1” “yes”, or “on” when a request is received via SSL. So, this function returns “https” if such a value is found, and “http” otherwise.

**wsgiref.util.request_uri(environ, include_query=True)**

Return the full request URI, optionally including the query string, using the algorithm found in the “URL Reconstruction” section of PEP 3333. If `include_query` is false, the query string is not included in the resulting URI.

**wsgiref.util.application_uri(environ)**

Similar to `request_uri()`, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.
wsgiref.util.shift_path_info(environ)

Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The environ dictionary is modified in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

If there are no remaining path segments in PATH_INFO, None is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls shift_path_info(), it will receive the string “bar”, and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

When PATH_INFO is just a “/”, this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /[x]/ when using this routine to do object traversal.

wsgiref.util.setup_testing_defaults(environ)

Update environ with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including
HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the PEP 3333-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage:

```python
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('
    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
        for key, value in environ.items()]
    return ret

    with make_server('', 8000, simple_app) as httpd:
        print("Serving on port 8000...")
        httpd.serve_forever()```
module also provides these miscellaneous utilities:

`wsgiref.util.is_hop_by_hop(header_name)`

Return true if ‘header_name’ is an HTTP/1.1 “Hop-by-Hop” header, as defined by [RFC 2616](https://www.rfc-editor.org/rfc/rfc2616).

`class wsgiref.util.FileWrapper(filelike, blksize=8192)`

A wrapper to convert a file-like object to an iterator. The resulting objects support both `__getitem__()` and `__iter__()` iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional `blksize` parameter will be repeatedly passed to the `filelike` object's `read()` method to obtain bytestrings to yield. When `read()` returns an empty bytestring, iteration is ended and is not resumable.

If `filelike` has a `close()` method, the returned object will also have a `close()` method, and it will invoke the `filelike` object's `close()` method when called.

Example usage:

```python
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object")
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```
wsgiref.headers – WSGI response header tools

This module provides a single class, Headers, for convenient manipulation of WSGI response headers using a mapping-like interface.

```python
class wsgiref.headers.Headers([headers])
```

Create a mapping-like object wrapping headers, which must be a list of header name/value tuples as described in PEP 3333. The default value of headers is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers’ existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, Headers objects do not raise an error when you try to get or delete a key that isn’t in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a Headers object is the same as the length of its
items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling bytes() on a Headers object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

**get_all(name)**

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

**add_header(name, value, **_params)**

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

*name* is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include
dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```python
h.add_header('content-disposition', 'attachment')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.
```

*Changed in version 3.5: headers parameter is optional.*
wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on http.server) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)

Create a new WSGI server listening on host and port, accepting connections for app. The return value is an instance of the supplied server_class, and will process requests using the specified handler_class. app must be a WSGI application object, as defined by PEP 3333.

Example usage:

```python
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```
wsgiref.simple_server.demo_app(environ, start_response)

This function is a small but complete WSGI application that returns a text page containing the message “Hello world!” and a list of the key/value pairs provided in the `environ` parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

```python
class wsgiref.simple_server.WSGIServer(server_address, RequestHandlerClass)

Create a WSGIServer instance. `server_address` should be a (host, port) tuple, and `RequestHandlerClass` should be the subclass of `http.server.BaseHTTPRequestHandler` that will be used to process requests.
```

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

WSGIServer is a subclass of `http.server.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. WSGIServer also provides these WSGI-specific methods:

```python
set_app(application)

Sets the callable `application` as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.
```

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`,
and the `get_app()` exists mainly for the benefit of request handler instances.

```python
class wsgiref.simple_server.WSGIRequestHandler(request, client_address, server)
    Create an HTTP handler for the given `request` (i.e. a socket), `client_address` (a `(host, port)` tuple), and `server` (`WSGIServer` instance).
```

You do not need to create instances of this class directly; they are automatically created as needed by `WSGIServer` objects. You can, however, subclass this class and supply it as a `handler_class` to the `make_server()` function. Some possibly relevant methods for overriding in subclasses:

**get.environ**

Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the `WSGIServer` object's `base.environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 3333.

**get.stderr**

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

**handle**

Process the HTTP request. The default implementation creates a handler instance using a `wsgiref.handlers` class to
implement the actual WSGI application interface.
wsgiref.validate — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using `wsgiref.validate`. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 3333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's “Python Paste” library.

`wsgiref.validate.validator(application)`

Wrap `application` and return a new WSGI application object. The returned application will forward all requests to the original `application`, and will check that both the `application` and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an `AssertionError` being raised; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don't override the error handling methods to do something else) will simply output
a message that an error has occurred, and dump the traceback to
\texttt{sys.stderr} or some other error stream.

This wrapper may also generate output using the \texttt{warnings}
module to indicate behaviors that are questionable but which may
not actually be prohibited by \texttt{PEP 3333}. Unless they are
suppressed using Python command-line options or the \texttt{warnings}
API, any such warnings will be written to \texttt{sys.stderr} (not\texttt{wsgi.errors}, unless they happen to be the same object).

Example usage:

\begin{verbatim}
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
\end{verbatim}
**wsgiref.handlers – server/gateway base classes**

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

### class wsgiref.handlers.CGIHandler

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of `BaseCGIHandler` that sets `wsgi.run_once` to true, `wsgi.multithread` to false, and `wsgi.multiprocess` to true, and always uses `sys` and `os` to obtain the necessary CGI streams and environment.

### class wsgiref.handlers.IISCGIHandler

A specialized alternative to `CGIHandler`, for use when deploying on Microsoft's IIS web server, without having set the config `allowPathInfo` option (IIS>=7) or `metabase allowPathInfoForScriptMappings` (IIS<7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.
IIS can be configured to pass the correct PATH_INFO, but this causes another bug where PATH_TRANSLATED is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the PATH_TRANSLATED bug. For this reason IIS<7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as CGIHandler, i.e., by calling IISCGIHandler().run(app), where app is the WSGI application object you wish to invoke.

New in version 3.2.

```
class wsgiref.handlers.BaseCGIHandler(stdin, stdout, stderr, environ, multithread=True, multiprocess=False)
```

Similar to CGIHandler, but instead of using the sys and os modules, the CGI environment and I/O streams are specified explicitly. The multithread and multiprocess values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of SimpleHandler intended for use with software other than HTTP “origin servers”. If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of SimpleHandler.

```
class wsgiref.handlers.SimpleHandler(stdin, stdout, stderr, environ, multithread=True, multiprocess=False)
```
Similar to `BaseCGIHandler`, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of `BaseCGIHandler`.

This class is a subclass of `BaseHandler`. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The `write()` method of `stdout` should write each chunk in full, like `io.BufferedIOBase`.

```python
class wsgiref.handlers.BaseHandler
```

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

`BaseHandler` instances have only one method intended for external use:

```python
run(app)
```

Run the specified WSGI application, `app`.

All of the other `BaseHandler` methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:
_write(data)
Buffer the bytes data for transmission to the client. It's okay if this method actually transmits the data; BaseHandler just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()
Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if _write() actually sends the data).

get_stdin()
Return an input stream object suitable for use as the wsgi.input of the request currently being processed.

get.Stderr()
Return an output stream object suitable for use as the wsgi.errors of the request currently being processed.

add_cgi_vars()
Insert CGI variables for the current request into the environ attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized BaseHandler subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread
The value to be used for the wsgi.multithread environment
variable. It defaults to true in `BaseHandler`, but may have a different default (or be set by the constructor) in the other subclasses.

**wsgi_multiprocess**
The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in `BaseHandler`, but may have a different default (or be set by the constructor) in the other subclasses.

**wsgi_run_once**
The value to be used for the `wsgi.run_once` environment variable. It defaults to false in `BaseHandler`, but `CGIHandler` sets it to true by default.

**os_environ**
The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that `wsgiref.handlers` was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

**server_software**
If the `origin_server` attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as `BaseCGIHandler` and `CGIHandler`) that are not HTTP origin servers.
Changed in version 3.3: The term “Python” is replaced with implementation specific term like “CPython”, “Jython” etc.

get_scheme()

Return the URL scheme being used for the current request. The default implementation uses the guess_scheme() function from wsgiref.util to guess whether the scheme should be “http” or “https”, based on the current request's environ variables.

setup_environ()

Set the environ attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the get_stdin(), get_stderr(), and add_cgi_vars() methods and the wsgi_file_wrapper attribute. It also inserts a SERVER_SOFTWARE key if not present, as long as the origin_server attribute is a true value and the server_software attribute is set.

Methods and attributes for customizing exception handling:

log_exception(exc_info)

Log the exc_info tuple in the server log. exc_info is a (type, value, traceback) tuple. The default implementation simply writes the traceback to the request's wsgi.errors stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default `log_exception()` method. If None, all frames are included.

**error_output**(environ, start_response)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to `start_response` when calling it (as described in the “Error Handling” section of PEP 3333).

The default implementation just uses the `error_status`, `error_headers`, and `error_body` attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it’s not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn’t include any.

**error_status**
The HTTP status used for error responses. This should be a status string as defined in PEP 3333; it defaults to a 500 code and message.

**error_headers**
The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as
described in PEP 3333. The default list just sets the content type to text/plain.

**error_body**

The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, “A server error occurred. Please contact the administrator.”

Methods and attributes for PEP 3333’s “Optional Platform-Specific File Handling” feature:

**wsgi_file_wrapper**

A wsgi.file_wrapper factory, or None. The default value of this attribute is the wsgiref.util.FileWrapper class.

**sendfile()**

Override to implement platform-specific file transmission. This method is called only if the application’s return value is an instance of the class specified by the wsgi_file_wrapper attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

**origin_server**

This attribute should be set to a true value if the handler’s _write() and _flush() are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special Status: header.

This attribute's default value is true in BaseHandler, but false
in BaseCGIHandler and CGIHandler.

**http_version**

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to "1.0".

**wsgiref.handlers.read_environ()**

Transcode CGI variables from `os.environ` to PEP 3333 “bytes in unicode” strings, returning a new dictionary. This function is used by CGIHandler and IISCGIHandler in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

*New in version 3.2.*
Examples

This is a working “Hello World” WSGI application:

```python
from wsgiref.simple_server import make_server

def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]
    start_response(status, headers)

    # The returned object is going to be printed
    return [b'Hello World']

with make_server('', 8000, hello_world_app) as httpd:
    print('Serving on port 8000...')

    # Serve until process is killed
    httpd.serve_forever()
```

urllib — URL handling modules

**Source code:** Lib/urllib/

urllib is a package that collects several modules for working with URLs:

- `urllib.request` for opening and reading URLs
- `urllib.error` containing the exceptions raised by `urllib.request`
- `urllib.parse` for parsing URLs
- `urllib.robotparser` for parsing `robots.txt` files
The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also: The Requests package is recommended for a higher-level HTTP client interface.

The `urllib.request` module defines the following functions:

```
urllib.request.urlopen(url, data=None, [timeout,]*, cafile=None, capath=None, cadefault=False, context=None)
```

Open the URL `url`, which can be either a string or a `Request` object.

`data` must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See `Request` for details.

The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.
If `context` is specified, it must be a `ssl.SSLContext` instance describing the various SSL options. See `HTTPSConnection` for more details.

The optional `cafile` and `capath` parameters specify a set of trusted CA certificates for HTTPS requests. `cafile` should point to a single file containing a bundle of CA certificates, whereas `capath` should point to a directory of hashed certificate files. More information can be found in `ssl.SSLContext.load_verify_locations()`.

The `cadefault` parameter is ignored.

This function always returns an object which can work as a context manager and has methods such as

- `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` — return the meta-information of the page, such as headers, in the form of an `email.message_from_string()` instance (see Quick Reference to HTTP Headers)
- `getcode()` — return the HTTP status code of the response.

For HTTP and HTTPS URLs, this function returns a `http.client.HTTPResponse` object slightly modified. In addition to the three new methods above, the `msg` attribute contains the same information as the `reason` attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for `HTTPResponse`.

For FTP, file, and data URLs and requests explicitly handled by legacy `URLopener` and `FancyURLopener` classes, this function returns a `urllib.response.addinfourl` object.
Raises **URLError** on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global `OpenerDirector` uses `UnknownHandler` to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), `ProxyHandler` is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; `urllib.request.urlopen()` corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using `ProxyHandler` objects.

*Changed in version 3.2:* `cafile` and `capath` were added.

*Changed in version 3.2:* HTTPS virtual hosts are now supported if possible (that is, if `ssl.HAS_SNI` is true).

*New in version 3.2:* `data` can be an iterable object.

*Changed in version 3.3:* `cadefault` was added.

*Changed in version 3.4.3:* `context` was added.

*Deprecated since version 3.6:* `cafile`, `capath` and `cadefault` are deprecated in favor of `context`. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system’s trusted CA certificates for you.
**urllib.request.install_opener**(opener)

Install an `OpenerDirector` instance as the default global opener. Installing an opener is only necessary if you want `urlopen` to use that opener; otherwise, simply call `OpenerDirector.open()` instead of `urlopen()`. The code does not check for a real `OpenerDirector`, and any class with the appropriate interface will work.

**urllib.request.build_opener**([`handler`, ...])

Return an `OpenerDirector` instance, which chains the handlers in the order given. `handlers` can be either instances of `BaseHandler`, or subclasses of `BaseHandler` (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the `handlers`, unless the `handlers` contain them, instances of them or subclasses of them: `ProxyHandler` (if proxy settings are detected), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

If the Python installation has SSL support (i.e., if the `ssl` module can be imported), `HTTPSHandler` will also be added.

A `BaseHandler` subclass may also change its `handler_order` attribute to modify its position in the handlers list.

**urllib.request.pathname2url**(path)

Convert the pathname `path` from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the `quote()` function.
url2pathname(path)
Convert the path component path from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses unquote() to decode path.

getproxies()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named <scheme>_proxy, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OS X System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

Note: If the environment variable REQUEST_METHOD is set, which usually indicates your script is running in a CGI environment, the environment variable HTTP_PROXY (uppercase _PROXY) will be ignored. This is because that variable can be injected by a client using the “Proxy:” HTTP header. If you need to use an HTTP proxy in a CGI environment, either use ProxyHandler explicitly, or make sure the variable name is in lowercase (or at least the _proxy suffix).

The following classes are provided:

class Request(url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)
This class is an abstraction of a URL request.

url should be a string containing a valid URL.
*data* must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, `HTTPHandler` will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in RFC 7230, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard `application/x-www-form-urlencoded` format. The `urllib.parse.urlencode()` function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

*headers* should be a dictionary, and will be treated as if `add_header()` was called with each key and value as arguments. This is often used to “spoof” the *User-Agent* header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while `urllib`’s default user agent string is "Python-urllib/2.6" (on Python 2.6).

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not `None`, `Content-Type: application/x-www-form-urlencoded` will be added as a default.
The final two arguments are only of interest for correct handling of third-party HTTP cookies:

*origin_req_host* should be the request-host of the origin transaction, as defined by RFC 2965. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

*unverifiable* should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

*method* should be a string that indicates the HTTP request method that will be used (e.g. 'HEAD'). If provided, its value is stored in the *method* attribute and is used by `get_method()`. The default is 'GET' if *data* is `None` or 'POST' otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

**Note:** The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.
Changed in version 3.3: `Request.method` argument is added to the Request class.

Changed in version 3.4: Default `Request.method` may be indicated at the class level.

Changed in version 3.6: Do not raise an error if the `Content-Length` has not been provided and `data` is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class `urllib.request.OpenerDirector`

The `OpenerDirector` class opens URLs via `BaseHandler`s chained together. It manages the chaining of handlers, and recovery from errors.

class `urllib.request.BaseHandler`

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class `urllib.request.HTTPDefaultErrorHandler`

A class which defines a default handler for HTTP error responses; all responses are turned into `HTTPError` exceptions.

class `urllib.request.HTTPRedirectHandler`

A class to handle redirections.

class
`urllib.request.HTTPCookieProcessor(cookiejar=None)`

A class to handle HTTP Cookies.

class `urllib.request.ProxyHandler(proxies=None)`

Cause requests to go through a proxy. If `proxies` is given, it must be a dictionary mapping protocol names to URLs of proxies. The
default is to read the list of proxies from the environment variables \texttt{<protocol>\_proxy}. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The no\_proxy environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with \texttt{:port} appended, for example \texttt{cern.ch,ncsa.uiuc.edu,some.host:8080}.

\textbf{Note:} \texttt{HTTP\_PROXY} will be ignored if a variable \texttt{REQUEST\_METHOD} is set; see the documentation on \texttt{getproxies()}.

class \texttt{urllib.request.HTTPPasswordMgr}

Keep a database of \texttt{(realm, uri) -> (user, password)} mappings.

class \texttt{urllib.request.HTTPPasswordMgrWithDefaultRealm}

Keep a database of \texttt{(realm, uri) -> (user, password)} mappings. A realm of \texttt{None} is considered a catch-all realm, which is searched if no other realm fits.

class \texttt{urllib.request.HTTPPasswordMgrWithPriorAuth}

A variant of \texttt{HTTPPasswordMgrWithDefaultRealm} that also has a database of \texttt{uri -> is\_authenticated} mappings. Can
be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a 401 response first.

New in version 3.5.

class urllib.request.AbstractBasicAuthHandler(password_mgr=None)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. If passwd_mgr also provides is_authenticated and update_authenticated methods (see HTTPPasswordMgrWithPriorAuth Objects), then the handler will use the is_authenticated result for a given URI to determine whether or not to send authentication credentials with the request. If is_authenticated returns True for the URI, credentials are sent. If is_authenticated is False, credentials are not sent, and then if a 401 response is received the request is re-sent with the authentication credentials. If authentication succeeds, update_authenticated is called to set is_authenticated True for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

New in version 3.5: Added is_authenticated support.

class urllib.request.HTTPBasicAuthHandler(password_mgr=None)

Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with
HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. HTTPBasicAuthHandler will raise a ValueError when presented with a wrong Authentication scheme.

class urllib.request.ProxyBasicAuthHandler(password_mgr=None)
    Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.AbstractDigestAuthHandler(password_mgr=None)
    This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.HTTPDigestAuthHandler(password_mgr=None)
    Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a ValueError when presented with an authentication scheme other than Digest or Basic.
Changed in version 3.3: Raise `ValueError` on unsupported Authentication Scheme.

class urllib.request.ProxyDigestAuthHandler(password_mgr=None)
Handle authentication with the proxy. `password_mgr`, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section `HTTPPasswordMgr Objects` for information on the interface that must be supported.

class urllib.request.HTTPHandler
A class to handle opening of HTTP URLs.

class urllib.request.HTTPSHandler(debuglevel=0, context=None, check_hostname=None)
A class to handle opening of HTTPS URLs. `context` and `check_hostname` have the same meaning as in `http.client.HTTPSConnection`.

Changed in version 3.2: `context` and `check_hostname` were added.

class urllib.request.FileHandler
Open local files.

class urllib.request.DataHandler
Open data URLs.

New in version 3.4.

class urllib.request.FTPHandler
Open FTP URLs.

class urllib.request.CacheFTPHandler
Open FTP URLs, keeping a cache of open FTP connections to
minimize delays.

class urllib.request.UnknownHandler
    A catch-all class to handle unknown URLs.

class urllib.request.HTTPErrorProcessor
    Process HTTP error responses.
Request Objects

The following methods describe Request’s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request. **full_url**
   The original URL passed to the constructor.

   *Changed in version 3.4.*

   Request.full_url is a property with setter, getter and a deleter. Getting **full_url** returns the original request URL with the fragment, if it was present.

Request. **type**
   The URI scheme.

Request. **host**
   The URI authority, typically a host, but may also contain a port separated by a colon.

Request. **origin_req_host**
   The original host for the request, without port.

Request. **selector**
   The URI path. If the Request uses a proxy, then selector will be the full URL that is passed to the proxy.

Request. **data**
   The entity body for the request, or None if not specified.

   *Changed in version 3.4:* Changing value of Request.data now
deletes “Content-Length” header if it was previously set or calculated.

Request.unverifiable
boolean, indicates whether the request is unverifiable as defined by RFC 2965.

Request.method
The HTTP request method to use. By default its value is None, which means that get_method() will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in get_method()) either by providing a default value by setting it at the class level in a Request subclass, or by passing a value in to the Request constructor via the method argument.

*New in version 3.3.*

*Changed in version 3.4:* A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.get_method()
Return a string indicating the HTTP request method. If Request.method is not None, return its value, otherwise return 'GET' if Request.data is None, or 'POST' if it’s not. This is only meaningful for HTTP requests.

*Changed in version 3.3:* get_method now looks at the value of Request.method.

Request.add_header(key, val)
Add another header to the request. Headers are currently ignored
by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the key collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.**add_unredirected_header**(key, header)
Add a header that will not be added to a redirected request.

Request.**has_header**(header)
Return whether the instance has the named header (checks both regular and unredirected).

Request.**remove_header**(header)
Remove named header from the request instance (both from regular and unredirected headers).

*New in version 3.4.*

Request.**get_full_url**()
Return the URL given in the constructor.

*Changed in version 3.4.*

Returns Request.**full_url**

Request.**set_proxy**(host, type)
Prepare the request by connecting to a proxy server. The host and type will replace those of the instance, and the instance’s selector will be the original URL given in the constructor.

Request.**get_header**(header_name, default=None)
Return the value of the given header. If the header is not present, return the default value.

Request.header_items()

Return a list of tuples (header_name, header_value) of the Request headers.

*Changed in version 3.4:* The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.
OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.**add_handler**(handler)

*handler* should be an instance of BaseHandler. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- **protocol_open()** — signal that the handler knows how to open *protocol* URLs.
- **http_error_type()** — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.
- **protocol_error()** — signal that the handler knows how to handle errors from (non-http) *protocol*.
- **protocol_request()** — signal that the handler knows how to pre-process *protocol* requests.
- **protocol_response()** — signal that the handler knows how to post-process *protocol* responses.

OpenerDirector.**open**(url, data=None[, timeout])

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of urlopen() (which simply calls the open() method on the currently installed global OpenerDirector). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

OpenerDirector.**error**(proto, *args)
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the http_error_*( ) methods of the handler classes.

Return values and exceptions raised are the same as those of urlopen().

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like protocol_request() has that method called to pre-process the request.

2. Handlers with a method named like protocol_open() are called to handle the request. This stage ends when a handler either returns a non-None value (ie. a response), or raises an exception (usually URLError). Exceptions are allowed to propagate.

In fact, the above algorithm is first tried for methods named default_open(). If all such methods return None, the algorithm is repeated for methods named like protocol_open(). If all such methods return None, the algorithm is repeated for methods named unknown_open().

Note that the implementation of these methods may involve calls of the parent OpenerDirector instance’s open() and error() methods.
3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.
BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler. `add_parent(director)`

Add a director as parent.

BaseHandler. `close()`

Remove any parents.

The following attribute and methods should only be used by classes derived from BaseHandler.

```
Note: The convention has been adopted that subclasses defining protocol_request() or protocol_response() methods are named *Processor; all others are named *Handler.
```

BaseHandler. `parent`

A valid OpenerDirector, which can be used to open using a different protocol, or handle errors.

BaseHandler. `default_open(req)`

This method is not defined in BaseHandler, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent OpenerDirector. It should return a file-like object as described in the return value of the `open()` of OpenerDirector, or None. It should raise URLError, unless a truly exceptional thing happens
(for example, `MemoryError` should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

**BaseHandler. `protocol_open(req)`**

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

**BaseHandler. `unknown_open(req)`**

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

**BaseHandler. `http_error_default(req, fp, code, msg, hdrs)`**

This method is *not* defined in `BaseHandler`, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the `OpenerDirector` getting the error, and should not normally be called in other circumstances.

`req` will be a `Request` object, `fp` will be a file-like object with the HTTP error body, `code` will be the three-digit code of the error, `msg`
will be the user-visible explanation of the code and `hdrs` will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of `urlopen()`.

**BaseHandler. `http_error_nnn`**

`nnn` should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code `nnn` occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

**BaseHandler. `protocol_request`**

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. `req` will be a `Request` object. The return value should be a `Request` object.

**BaseHandler. `protocol_response`**

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. `req` will be a `Request` object. `response` will be
an object implementing the same interface as the return value of \texttt{urlopen()}. The return value should implement the same interface as the return value of \texttt{urlopen()}. 
HTTPRedirectHandler Objects

**Note:** Some HTTP redirections require action from this module's client code. If this is the case, HTTPError is raised. See RFC 2616 for details of the precise meanings of the various redirection codes.

An HTTPError exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

**HTTPRedirectHandler.** `redirect_request(req, fp, code, msg, hdrs, newurl)`

Return a Request or None in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new Request to allow `http_error_30*()` to perform the redirect to `newurl`. Otherwise, raise HTTPError if no other handler should try to handle this URL, or return None if you can’t but another handler might.

**Note:** The default implementation of this method does not strictly follow RFC 2616, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

**HTTPRedirectHandler.** `http_error_301(req, fp, code, msg, hdrs)`

Redirect to the Location: or URI: URL. This method is called by
the parent **OpenerDirector** when getting an HTTP ‘moved permanently’ response.

HTTPRedirectHandler.**http_error_302**(req, fp, code, msg, hdrs)

The same as **http_error_301()**, but called for the ‘found’ response.

HTTPRedirectHandler.**http_error_303**(req, fp, code, msg, hdrs)

The same as **http_error_301()**, but called for the ‘see other’ response.

HTTPRedirectHandler.**http_error_307**(req, fp, code, msg, hdrs)

The same as **http_error_301()**, but called for the ‘temporary redirect’ response.
HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.cookiejar

The http.cookiejar.CookieJar in which cookies are stored.
ProxyHandler Objects

ProxyHandler. **protocol_open**(request)

The **ProxyHandler** will have a method **protocol_open()** for every protocol which has a proxy in the **proxies** dictionary given in the constructor. The method will modify requests to go through the proxy, by calling **request.set_proxy()**, and call the next handler in the chain to actually execute the protocol.
HTTPPasswordMgr Objects

These methods are available on HTTPPasswordMgr and HTTPPasswordMgrWithDefaultRealm objects.

HTTPPasswordMgr.add_password(realm, uri, user, passwd)

uri can be either a single URI, or a sequence of URIs. realm, user and passwd must be strings. This causes (user, passwd) to be used as authentication tokens when authentication for realm and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.find_user_password(realm, authuri)

Get user/password for given realm and URI, if any. This method will return (None, None) if there is no matching user/password.

For HTTPPasswordMgrWithDefaultRealm objects, the realm None will be searched if the given realm has no matching user/password.
HTTPPasswordMgrWithPriorAuth Objects

This password manager extends HTTPPasswordMgrWithDefaultRealm to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.add_password(realm, uri, user, passwd, is_authenticated=False)

realm, uri, user, passwd are as for HTTPPasswordMgr.add_password(). is_authenticated sets the initial value of the is_authenticated flag for the given URI or list of URIs. If is_authenticated is specified as True, realm is ignored.

HTTPPasswordMgr.find_user_password(realm, authuri)

Same as for HTTPPasswordMgrWithDefaultRealm objects

HTTPPasswordMgrWithPriorAuth.update_authenticated(uri, is_authenticated=False)

Update the is_authenticated flag for the given uri or list of URIs.

HTTPPasswordMgrWithPriorAuth.is_authenticated(self, authuri)

Returns the current state of the is_authenticated flag for the given URI.
AbstractBasicAuthHandler Objects

`AbstractBasicAuthHandler.http_error_auth_reqed(authreq, host, req, headers)`

Handle an authentication request by getting a user/password pair, and re-trying the request. `authreq` should be the name of the header where the information about the realm is included in the request, `host` specifies the URL and path to authenticate for, `req` should be the (failed) `Request` object, and `headers` should be the error headers.

`host` is either an authority (e.g. "python.org") or a URL containing an authority component (e.g. "http://python.org/"). In either case, the authority must not contain a userinfo component (so, "python.org" and "python.org:80" are fine, "joe:password@python.org" is not).
HTTPBasicAuthHandler Objects

HTTPBasicAuthHandler.\texttt{http\_error\_401}(req, fp, code, msg, hdrs)

- Retry the request with authentication information, if available.
ProxyBasicAuthHandler Objects

ProxyBasicAuthHandler.\texttt{http\_error\_407}(\textit{req}, \textit{fp}, \textit{code}, \textit{msg}, \textit{hdrs})

Retry the request with authentication information, if available.
AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

*authreq* should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) *Request* object, and *headers* should be the error headers.
HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.http_error_401(req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.\texttt{http\_error\_407}(req, fp, code, msg, hdrs)

Retry the request with authentication information, if available.
HTTPHandler Objects

HTTPHandler.\texttt{http\_open}(\textit{req})

Send an HTTP request, which can be either GET or POST, depending on \textit{req.has\_data()}. 
HTTPSHandler Objects

HTTPSHandler. `https_open(req)`

Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.
FileHandler Objects

FileHandler.\texttt{file\_open}(\texttt{req})

Open the file locally, if there is no host name, or the host name is 'localhost'.

\textit{Changed in version 3.2}: This method is applicable only for local hostnames. When a remote hostname is given, an \texttt{URLError} is raised.
DataHandler Objects

DataHandler.\texttt{data\_open}(\textit{req})

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in RFC 2397. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don’t mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an \texttt{ValueError} in that case.
FTPHandler Objects

FTPHandler..ftp_open(req)

Open the FTP file indicated by `req`. The login is always done with empty username and password.
CacheFTPHandler Objects

CacheFTPHandler objects are FTPHandler objects with the following additional methods:

CacheFTPHandler.setTimeout($t$)
  Set timeout of connections to $t$ seconds.

CacheFTPHandler.setMaxConns($m$)
  Set maximum number of cached connections to $m$. 
UnknownHandler Objects

UnknownHandler. **unknown_open()**

Raise a **URLError** exception.
HTTPErrorProcessor Objects

HTTPErrorProcessor. **http_response**(request, response)
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `protocol_error_code()` handler methods, via `OpenerDirector.error()`.
Eventually, `HTTPDefaultErrorHandler` will raise an **HTTPError** if no other handler handles the error.

HTTPErrorProcessor. **https_response**(request, response)
Process HTTPS error responses.

The behavior is same as **http_response**().
Examples

In addition to the examples below, more examples are given in HOWTO Fetch Internet Resources Using The urllib Package.

This example gets the python.org main page and displays the first 300 bytes of it.

```python
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/......
    print(f.read(300))
...
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses utf-8 encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```python
>>> with urllib.request.urlopen('http://www.python.org/......
    print(f.read(100).decode('utf-8'))
```
It is also possible to achieve the same result without using the context manager approach.

```python
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/
... print(f.read(100).decode('utf-8'))
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```python
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi
... data=b'This data is passed to stdin of the CGI'
... with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
```

The code for the sample CGI used in the above example is:

```python
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' %
```

Here is an example of doing a PUT request using `Request`:
Use of Basic HTTP Authentication:

```python
import urllib.request

# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                           uri='https://mahler:8092/site-updates.py',
                           user='klem',
                           passwd='kadidd!ehopper')

opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urllib.request.urlopen(opener)

urllib.request.urlopen('http://www.example.com/login.html')
```

`build_opener()` provides many handlers by default, including a `ProxyHandler`. By default, `ProxyHandler` uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP proxy’s URL.

This example replaces the default `ProxyHandler` with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with `ProxyBasicAuthHandler`.
proxy_handler = urllib.request.ProxyHandler({'http':
  proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
  proxy_auth_handler.add_password('realm', 'host', 'user', 'pass')

opener = urllib.request.build_opener(proxy_handler,
  # This time, rather than install the OpenerDirector,
  opener.open('http://www.example.com/login.html')

Adding HTTP headers:

Use the headers argument to the Request constructor, or:

```python
import urllib.request
req = urllib.request.Request('http://www.example.com/',
  headers = {'Referer': 'http://www.python.org/'}
  # Customize the default User-Agent header value:
  req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . . )
  r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a User-Agent header to every Request. To change this:

```python
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (Content-Length, Content-Type and Host) are added when the Request is passed to urlopen() (or OpenerDirector.open()).

Here is an example session that uses the GET method to retrieve a URL containing parameters:
The following example uses the POST method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```python
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr") as f:
...    print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```python
>>> import urllib.request
>>> proxies = { 'http': 'http://proxy.example.com:8080/' }
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...    f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:
```python
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...    f.read().decode('utf-8')
...
Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.urlretrieve(url, filename=None, reporthook=None, data=None)`

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where `filename` is the local file name under which the object can be found, and `headers` is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```python
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve
>>> html = open(local_filename)
```
If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is GET). The `data` argument must be a bytes object in standard `application/x-www-form-urlencoded` format; see the `urllib.parse.urlencode()` function.

`urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a `Content-Length` header). This can occur, for example, when the download is interrupted.

The `Content-Length` is treated as a lower bound: if there’s more data to read, `urlretrieve` reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no `Content-Length` header was supplied, `urlretrieve` can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.urlcleanup()`  
Cleans up temporary files that may have been left behind by previous calls to `urlretrieve()`.

`class urllib.request.URLopener(proxies=None, **x509)`  
*Deprecated since version 3.3.*
Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than http:, ftp:, or file:, you probably want to use FancyURLopener.

By default, the URLopener class sends a User-Agent header of urllib/VVV, where VVV is the urllib version number. Applications can define their own User-Agent header by subclassing URLopener or FancyURLopener and setting the class attribute version to an appropriate string value in the subclass definition.

The optional proxies parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is None, in which case environmental proxy settings will be used if present, as discussed in the definition of urlopen(), above.

Additional keyword parameters, collected in x509, may be used for authentication of the client when using the https: scheme. The keywords key_file and cert_file are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an OSError exception if the server returns an error code.

**open**(fullurl, data=None)

Open fullurl using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, open_unknown() is called. The data argument has the same meaning as the data argument of urlopen().
**open unknown** *(fullurl, data=None)*

Overridable interface to open unknown URL types.

**retrieve** *(url, filename=None, reporthook=None, data=None)*

Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or *None* (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the *http:* scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

**version**

Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.
class urllib.request.FancyURLopener(...)

Deprecated since version 3.3.

FancyURLopener subclasses URLopener providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the Location header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the maxtries attribute, which defaults to 10.

For all other response codes, the method http_error_default() is called which you can override in subclasses to handle the error appropriately.

Note: According to the letter of RFC 2616, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and urllib reproduces this behaviour.

The parameters to the constructor are the same as those for URLopener.

Note: When performing basic authentication, a FancyURLopener instance calls its prompt_user_passwd() method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The FancyURLopener class offers one additional method that
should be overloaded to provide the appropriate behavior:

**prompt_user_passwd**(*host, realm*)

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, *(user, password)*, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.
urllib.request Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  *Changed in version 3.4: Added support for data URLs.*

- The caching feature of urllretrive() has been disabled until someone finds the time to hack proper processing of Expiration time headers.

- There should be a function to query whether a particular URL is in the cache.

- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

- The urlopen() and urllretrive() functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.

- The data returned by urlopen() or urllretrive() is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the Content-Type header. If the returned data is HTML, you can use the module html.parser to parse it.

- The code handling the FTP protocol cannot differentiate between
a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module, subclassing `FancyURLopener`, or changing `_urloper` to meet your needs.
The `urllib.response` module defines functions and classes which define a minimal file-like interface, including `read()` and `readline()`. The typical response object is an `addinfourl` instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the `urllib.request` module.
urllib.parse — Parse URLs into components

Source code: Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais, ws, wss.

The urllib.parse module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.
URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

`urllib.parse.urlparse(urlstring, scheme='', allow_fragments=True)`

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the `path` component, which is retained if present. For example:

```python
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o  # doctest: +NORMALIZE_WHITESPACE
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in RFC 1808, `urlparse` recognizes a netloc only if it is properly introduced by `//`. Otherwise the input is presumed to be a relative URL and thus to start with a
path component.

```python
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html', params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='', query='', fragment='')
```

The `scheme` argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as `urlstring`, except that the default value '' is always allowed, and is automatically converted to b'' if appropriate.

If the `allow_fragments` argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and fragment is set to the empty string in the return value.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td><code>scheme</code> parameter</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location</td>
<td>empty string</td>
</tr>
<tr>
<td>part</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>------------</td>
<td>-----</td>
<td>-------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>params</td>
<td>3</td>
<td>Parameters for last path element</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>4</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>5</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

Reading the port attribute will raise a `ValueError` if an invalid port is specified in the URL. See section Structured Parse Results for more information on the result object.

Unmatched square brackets in the netloc attribute will raise a `ValueError`.

Characters in the netloc attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

*Changed in version 3.2:* Added IPv6 URL parsing capabilities.

*Changed in version 3.3:* The fragment is now parsed for all URL schemes (unless `allow_fragment` is false), in accordance with RFC 3986. Previously, a whitelist of schemes that support fragments existed.
Changed in version 3.6: Out-of-range port numbers now raise \texttt{ValueError}, instead of returning \texttt{None}.

Changed in version 3.7.3: Characters that affect netloc parsing under NFKC normalization will now raise \texttt{ValueError}.

\texttt{urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None)}

Parse a query string given as a string argument (data of type \texttt{application/x-www-form-urlencoded}). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument \texttt{keep_blank_values} is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument \texttt{strict_parsing} is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a \texttt{ValueError} exception.

The optional \texttt{encoding} and \texttt{errors} parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the \texttt{bytes.decode()} method.

The optional argument \texttt{max_num_fields} is the maximum number of fields to read. If set, then throws a \texttt{ValueError} if there are more than \texttt{max_num_fields} fields read.
Use the `urllib.parse.urlencode()` function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

*Changed in version 3.2:* Add `encoding` and `errors` parameters.

*Changed in version 3.7.2:* Added `max_num_fields` parameter.

```python
urllib.parse.parse_qsl(qs, keep_blank_values=False,
strict_parsing=False, encoding='utf-8', errors='replace',
max_num_fields=None)
```

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional `encoding` and `errors` parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument `max_num_fields` is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than `max_num_fields` fields read.
Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

*Changed in version 3.2:* Add *encoding* and *errors* parameters.

*Changed in version 3.7.2:* Added *max_num_fields* parameter.

`urllib.parse.urlunparse(parts)`

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.urlsplit(urlstring, scheme='', allow_fragments=True)`

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see RFC 2396) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td><code>scheme</code> parameter</td>
</tr>
<tr>
<td>Component</td>
<td>Number</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>3</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>4</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section [Structured Parse Results](#) for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

*Changed in version 3.6:* Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

*Changed in version 3.7.3:* Characters that affect `netloc` parsing under NFKC normalization will now raise `ValueError`.

```
urllib.parse.urlunsplit(parts)
```
Combine the elements of a tuple as returned by `urllib.parse.urlsplit()` into a complete URL as a string. The `parts` argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

**urllib.parse.urljoin(base, url, allow_fragments=True)**

Construct a full (“absolute”) URL by combining a “base URL” (`base`) with another URL (`url`). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```python
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'http://www.cwi.nl/%7Eguido/FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The `allow_fragments` argument has the same meaning and default as for `urlparse()`.

**Note:** If `url` is an absolute URL (that is, starting with // or scheme://), the `url`'s host name and/or scheme will be present in the result. For example:

```python
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', '
... 'http://www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the `url` with `urlsplit()` and `urlunsplit()`, removing possible scheme
and netloc parts.

*Changed in version 3.5:* Behaviour updated to match the semantics defined in [RFC 3986](https://tools.ietf.org/html/rfc3986).

```python
urllib.parse.urldefrag(url)
```

If `url` contains a fragment identifier, return a modified version of `url` with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in `url`, return `url` unmodified and an empty string.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>0</td>
<td>URL with no fragment</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>1</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
</tbody>
</table>

See [section Structured Parse Results](https://docs.python.org/3/library/urllib.parse.html#structured-parse-results) for more information on the result object.

*Changed in version 3.2:* Result is a structured object rather than a simple 2-tuple.
Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an `encode()` method (when the result contains `str` data) or a `decode()` method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is `ascii` rather than `utf-8`). Each produces a value of a corresponding type that contains either `bytes` data (for `encode()` methods) or `str` data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.
The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

*Changed in version 3.2:* URL parsing functions now accept ASCII encoded byte sequences
Structured Parse Results

The result objects from the `urlparse()`, `urlsplit()` and `urldefrag()` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

`urllib.parse.SplitResult.geturl()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For `urldefrag()` results, only empty fragment identifiers will be removed. For `urlsplit()` and `urlparse()` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```python
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on `str` objects:
class urllib.parse.**DefragResult**(url, fragment)
Concrete class for `urldefrag()` results containing `str` data. The `encode()` method returns a `DefragResultBytes` instance.

*New in version 3.2.*

class urllib.parse.**ParseResult**(scheme, netloc, path, params, query, fragment)
Concrete class for `urlparse()` results containing `str` data. The `encode()` method returns a `ParseResultBytes` instance.

class urllib.parse.**SplitResult**(scheme, netloc, path, query, fragment)
Concrete class for `urlsplit()` results containing `str` data. The `encode()` method returns a `SplitResultBytes` instance.

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects:

class urllib.parse.**DefragResultBytes**(url, fragment)
Concrete class for `urldefrag()` results containing `bytes` data. The `decode()` method returns a `DefragResult` instance.

*New in version 3.2.*

class urllib.parse.**ParseResultBytes**(scheme, netloc, path, params, query, fragment)
Concrete class for `urlparse()` results containing `bytes` data. The `decode()` method returns a `ParseResult` instance.

*New in version 3.2.*
class urllib.parse.SplitResultBytes(scheme, netloc, path, query, fragment)

Concrete class for urlsplit() results containing bytes data. The decode() method returns a SplitResult instance.

New in version 3.2.
URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

```
urllib.parse.quote(string, safe='/', encoding=None, errors=None)
```

Replace special characters in `string` using the `%xx` escape. Letters, digits, and the characters `'_-.~'` are never quoted. By default, this function is intended for quoting the path section of URL. The optional `safe` parameter specifies additional ASCII characters that should not be quoted — its default value is `'/'`.

`string` may be either a `str` or a `bytes`.

*Changed in version 3.7:* Moved from RFC 2396 to RFC 3986 for quoting URL strings. “~” is now included in the set of reserved characters.

The optional `encoding` and `errors` parameters specify how to deal with non-ASCII characters, as accepted by the `str.encode()` method. `encoding` defaults to `'utf-8'`. `errors` defaults to `'strict'`, meaning unsupported characters raise a `UnicodeEncodeError`. `encoding` and `errors` must not be supplied if `string` is a `bytes`, or a `TypeError` is raised.

Note that `quote(string, safe, encoding, errors)` is
equivalent to
quote_from_bytes(string.encode(encoding, errors), safe).

Example: quote('/El Niño/') yields '/El%20Ni%C3%B1o/'.

urllib.parse.quote_plus(string, safe='', encoding=None, errors=None)

Like quote(), but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in safe. It also does not have safe default to '/'.

Example: quote_plus('/El Niño/') yields '%2FE1+Ni%C3%B1o%2F'.

urllib.parse.quote_from_bytes(bytes, safe='/')

Like quote(), but accepts a bytes object rather than a str, and does not perform string-to-bytes encoding.

Example: quote_from_bytes(b'a&\xef') yields 'a%26%EF'.

urllib.parse.unquote(string, encoding='utf-8', errors='replace')

Replace %xx escapes by their single-character equivalent. The optional encoding and errors parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the bytes.decode() method.

string must be a str.

encoding defaults to 'utf-8'. errors defaults to 'replace', meaning invalid sequences are replaced by a placeholder character.
Example: `unquote('/El%20Ni%C3%B1o/')` yields `/El Niño/`.

```python
urllib.parse.unquote_plus(string, encoding='utf-8', errors='replace')
```

Like `unquote()`, but also replace plus signs by spaces, as required for unquoting HTML form values.

`string` must be a `str`.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `/El Niño/`.

```python
urllib.parse.unquote_to_bytes(string)
```

Replace `%xx` escapes by their single-octet equivalent, and return a `bytes` object.

`string` may be either a `str` or a `bytes`.

If it is a `str`, unescaped non-ASCII characters in `string` are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields b'a&\xef'.

```python
urllib.parse.urlencode(query, doseq=False, safe='', encoding=None, errors=None, quote_via=quote_plus)
```

Convert a mapping object or a sequence of two-element tuples, which may contain `str` or `bytes` objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a `data` for POST operation with the `urlopen()` function, then it should be encoded to bytes, otherwise it would result in a `TypeError`.

The resulting string is a series of `key=value` pairs separated by
'&' characters, where both key and value are quoted using the quote_via function. By default, quote_plus() is used to quote the values, which means spaces are quoted as a '+1' character and '!' characters are encoded as %2F, which follows the standard for GET requests (application/x-www-form-urlencoded). An alternate function that can be passed as quote_via is quote(), which will encode spaces as %20 and not encode '!' characters. For maximum control of what is quoted, use quote and specify a value for safe.

When a sequence of two-element tuples is used as the query argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter doseq is evaluates to True, individual key=value pairs separated by '&1' are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The safe, encoding, and errors parameters are passed down to quote_via (the encoding and errors parameters are only passed when a query element is a str).

To reverse this encoding process, parse_qs() and parse_qsl() are provided in this module to parse query strings into Python data structures.

Refer to urllib examples to find out how urlencode method can be used for generating query string for a URL or data for POST.

Changed in version 3.2: Query parameter supports bytes and string objects.
New in version 3.5: quote_via parameter.

See also:

**RFC 3986 - Uniform Resource Identifiers**
This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

**RFC 2732 - Format for Literal IPv6 Addresses in URL’s.**
This specifies the parsing requirements of IPv6 URLs.

**RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax**
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

**RFC 2368 - The mailto URL scheme.**
Parsing requirements for mailto URL schemes.

**RFC 1808 - Relative Uniform Resource Locators**
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of “Abnormal Examples” which govern the treatment of border cases.

**RFC 1738 - Uniform Resource Locators (URL)**
This specifies the formal syntax and semantics of absolute URLs.
The `urllib.error` module defines the exception classes for exceptions raised by `urllib.request`. The base exception class is `URLError`.

The following exceptions are raised by `urllib.error` as appropriate:

- **exception `urllib.error.URLError`**
  - The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of `OSError`.
  
  - **reason**
    - The reason for this error. It can be a message string or another exception instance.

  *Changed in version 3.3:* `URLError` has been made a subclass of `OSError` instead of `IOError`.

- **exception `urllib.error.HTTPError`**
  - Though being an exception (a subclass of `URLError`), an `HTTPError` can also function as a non-exceptional file-like return value (the same thing that `urlopen()` returns). This is useful when handling exotic HTTP errors, such as requests for authentication.
**code**

An HTTP status code as defined in [RFC 2616](https://www.ietf.org/rfc/rfc2616.txt). This numeric value corresponds to a value found in the dictionary of codes as found in `http.server.BaseHTTPRequestHandler.responses`.

**reason**

This is usually a string explaining the reason for this error.

**headers**

The HTTP response headers for the HTTP request that caused the `HTTPError`.

*New in version 3.4.*

**exception `urllib.error.ContentTooShortError`(msg, content)**

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the `Content-Length` header). The `content` attribute stores the downloaded (and supposedly truncated) data.
The `urllib.robotparser` module provides a single class, `RobotFileParser`, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see [http://www.robotstxt.org/orig.html](http://www.robotstxt.org/orig.html).

```python
class urllib.robotparser.RobotFileParser(url='')
```

This class provides methods to read, parse and answer questions about the `robots.txt` file at `url`.

```python
set_url(url)
```

Sets the URL referring to a `robots.txt` file.

```python
read()
```

Reads the `robots.txt` URL and feeds it to the parser.

```python
parse(lines)
```

Parses the lines argument.

```python
can_fetch(useragent, url)
```

Returns `True` if the `useragent` is allowed to fetch the `url` according to the rules contained in the parsed `robots.txt` file.

```python
mtime()
```
Returns the time the `robots.txt` file was last fetched. This is useful for long-running web spiders that need to check for new `robots.txt` files periodically.

`modified()`
Sets the time the `robots.txt` file was last fetched to the current time.

`crawl_delay(useragent)`
Returns the value of the Crawl-delay parameter from `robots.txt` for the `useragent` in question. If there is no such parameter or it doesn't apply to the `useragent` specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

*New in version 3.6.*

`request_rate(useragent)`
Returns the contents of the Request-rate parameter from `robots.txt` as a named tuple `RequestRate(requests, seconds)`. If there is no such parameter or it doesn't apply to the `useragent` specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

*New in version 3.6.*

The following example demonstrates basic use of the `RobotFileParser` class:

```python
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musical.com/robots.txt")
>>> rp.read()
```
```python
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```
http — HTTP modules

Source code: Lib/http/__init__.py

http is a package that collects several modules for working with the HyperText Transfer Protocol:

- http.client is a low-level HTTP protocol client; for high-level URL opening use urllib.request
- http.server contains basic HTTP server classes based on socketserver
- http.cookies has utilities for implementing state management with cookies
- http.cookiejar provides persistence of cookies

http is also a module that defines a number of HTTP status codes and associated messages through the http.HTTPStatus enum:

class http.HTTPStatus

    New in version 3.5.

    A subclass of enum.IntEnum that defines a set of HTTP status codes, reason phrases and long descriptions written in English.

Usage:

```python
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> http.HTTPStatus.OK.value
```
200

>>> HTTPStatus.OK.phrase
'OK'

>>> HTTPStatus.OK.description
'Request fulfilled, document follows'

>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
HTTP status codes

Supported, IANA-registered status codes available in `http.HTTPStatus` are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Enum Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>CONTINUE</td>
<td>HTTP/1.1 [RFC 7231], Section 6.2.1</td>
</tr>
<tr>
<td>101</td>
<td>SWITCHING_PROTOCOLS</td>
<td>HTTP/1.1 [RFC 7231], Section 6.2.2</td>
</tr>
<tr>
<td>102</td>
<td>PROCESSING</td>
<td>WebDAV [RFC 2518], Section 10.1</td>
</tr>
<tr>
<td>200</td>
<td>OK</td>
<td>HTTP/1.1 [RFC 7231], Section 6.3.1</td>
</tr>
<tr>
<td>201</td>
<td>CREATED</td>
<td>HTTP/1.1 [RFC 7231], Section 6.3.2</td>
</tr>
<tr>
<td>202</td>
<td>ACCEPTED</td>
<td>HTTP/1.1 [RFC 7231], Section 6.3.3</td>
</tr>
<tr>
<td>203</td>
<td>NON_AUTHORITATIVE_INFORMATION</td>
<td>HTTP/1.1 [RFC 7231], Section 6.3.4</td>
</tr>
<tr>
<td>204</td>
<td>NO_CONTENT</td>
<td>HTTP/1.1 [RFC 7231], Section 6.3.5</td>
</tr>
<tr>
<td>Status Code</td>
<td>Status Word</td>
<td>Protocol/Specification</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>205</td>
<td>RESET_CONTENT</td>
<td>HTTP/1.1 RFC 7231, Section 6.3.6</td>
</tr>
<tr>
<td>206</td>
<td>PARTIAL_CONTENT</td>
<td>HTTP/1.1 RFC 7233, Section 4.1</td>
</tr>
<tr>
<td>207</td>
<td>MULTI_STATUS</td>
<td>WebDAV RFC 4918, Section 11.1</td>
</tr>
<tr>
<td>208</td>
<td>ALREADY_REPORTED</td>
<td>WebDAV Binding Extensions RFC 5842, Section 7.1 (Experimental)</td>
</tr>
<tr>
<td>226</td>
<td>IM_USED</td>
<td>Delta Encoding in HTTP RFC 3229, Section 10.4.1</td>
</tr>
<tr>
<td>300</td>
<td>MULTIPLE_CHOICES</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.1</td>
</tr>
<tr>
<td>301</td>
<td>MOVED_PERMANENTLY</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.2</td>
</tr>
<tr>
<td>302</td>
<td>FOUND</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.3</td>
</tr>
<tr>
<td>303</td>
<td>SEE_OTHER</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.4</td>
</tr>
<tr>
<td>Status Code</td>
<td>Reason Phrase</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>304</td>
<td>NOT_MODIFIED</td>
<td>7232, Section 4.1</td>
</tr>
<tr>
<td>305</td>
<td>USE_PROXY</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.5</td>
</tr>
<tr>
<td>307</td>
<td>TEMPORARY_REDIRECT</td>
<td>HTTP/1.1 RFC 7231, Section 6.4.7</td>
</tr>
<tr>
<td>308</td>
<td>PERMANENT_REDIRECT</td>
<td>Permanent Redirect RFC 7238, Section 3 (Experimental)</td>
</tr>
<tr>
<td>400</td>
<td>BAD_REQUEST</td>
<td>HTTP/1.1 RFC 7231, Section 6.5.1</td>
</tr>
<tr>
<td>401</td>
<td>UNAUTHORIZED</td>
<td>HTTP/1.1 Authentication RFC 7235, Section 3.1</td>
</tr>
<tr>
<td>402</td>
<td>PAYMENT_REQUIRED</td>
<td>HTTP/1.1 RFC 7231, Section 6.5.2</td>
</tr>
<tr>
<td>403</td>
<td>FORBIDDEN</td>
<td>HTTP/1.1 RFC 7231, Section 6.5.3</td>
</tr>
<tr>
<td>404</td>
<td>NOT_FOUND</td>
<td>HTTP/1.1 RFC 7231, Section 6.5.4</td>
</tr>
<tr>
<td>405</td>
<td>METHOD_NOT_ALLOWED</td>
<td>HTTP/1.1 RFC 7231, Section 6.5.5</td>
</tr>
<tr>
<td>Status Code</td>
<td>Reason Phrase</td>
<td>HTTP/1.1 RFC</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>406</td>
<td>NOT_ACCEPTABLE</td>
<td>7231</td>
</tr>
<tr>
<td>407</td>
<td>PROXY_AUTHENTICATION_REQUIRED</td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>REQUEST_TIMEOUT</td>
<td>7231</td>
</tr>
<tr>
<td>409</td>
<td>CONFLICT</td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>GONE</td>
<td>7231</td>
</tr>
<tr>
<td>411</td>
<td>LENGTH_REQUIRED</td>
<td>7231</td>
</tr>
<tr>
<td>412</td>
<td>PRECONDITION_FAILED</td>
<td>7232</td>
</tr>
<tr>
<td>413</td>
<td>REQUEST_ENTITY_TOO_LARGE</td>
<td>7231</td>
</tr>
<tr>
<td>414</td>
<td>REQUEST_URI_TOO_LONG</td>
<td>7231</td>
</tr>
<tr>
<td>415</td>
<td>UNSUPPORTED_MEDIA_TYPE</td>
<td>7231</td>
</tr>
<tr>
<td>Status Code</td>
<td>Reason Phrase</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>416</td>
<td>REQUESTED_RANGE_NOT_SATISFIABLE</td>
<td>Range Requests <a href="https://tools.ietf.org/html/rfc7233">RFC 7233</a>, Section 4.4</td>
</tr>
<tr>
<td>417</td>
<td>EXPECTATION_FAILED</td>
<td>HTTP/1.1 <a href="https://tools.ietf.org/html/rfc7231">RFC 7231</a>, Section 6.5.14</td>
</tr>
<tr>
<td>421</td>
<td>MISDIRECTED_REQUEST</td>
<td>HTTP/2 <a href="https://tools.ietf.org/html/rfc7540">RFC 7540</a>, Section 9.1.2</td>
</tr>
<tr>
<td>422</td>
<td>UNPROCESSABLE_ENTITY</td>
<td>WebDAV <a href="https://tools.ietf.org/html/rfc4918">RFC 4918</a>, Section 11.2</td>
</tr>
<tr>
<td>423</td>
<td>LOCKED</td>
<td>WebDAV <a href="https://tools.ietf.org/html/rfc4918">RFC 4918</a>, Section 11.3</td>
</tr>
<tr>
<td>424</td>
<td>FAILED_DEPENDENCY</td>
<td>WebDAV <a href="https://tools.ietf.org/html/rfc4918">RFC 4918</a>, Section 11.4</td>
</tr>
<tr>
<td>426</td>
<td>UPGRADE_REQUIRED</td>
<td>HTTP/1.1 <a href="https://tools.ietf.org/html/rfc7231">RFC 7231</a>, Section 6.5.15</td>
</tr>
<tr>
<td>428</td>
<td>PRECONDITION_REQUIRED</td>
<td>Additional HTTP Status Codes <a href="https://tools.ietf.org/html/rfc6585">RFC 6585</a></td>
</tr>
<tr>
<td>429</td>
<td>TOO_MANY_REQUESTS</td>
<td>Additional HTTP Status Codes <a href="https://tools.ietf.org/html/rfc6585">RFC 6585</a></td>
</tr>
<tr>
<td>Code</td>
<td>Status Code</td>
<td>Status Message</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>431</td>
<td>REQUEST_HEADER_FIELDS_TOO_LARGE</td>
<td>HTTP Status Codes</td>
</tr>
<tr>
<td>500</td>
<td>INTERNAL_SERVER_ERROR</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>501</td>
<td>NOT_IMPLEMENTED</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>502</td>
<td>BAD_GATEWAY</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>503</td>
<td>SERVICE_UNAVAILABLE</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>504</td>
<td>GATEWAY_TIMEOUT</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>505</td>
<td>HTTP_VERSION_NOT_SUPPORTED</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>506</td>
<td>VARIANT_ALSO_NEGOTIATES</td>
<td>Transparent Content Negotiation in HTTP</td>
</tr>
<tr>
<td>507</td>
<td>INSUFFICIENT_STORAGE</td>
<td>WebDAV</td>
</tr>
</tbody>
</table>

WebDAV
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>508</td>
<td>LOOP_DETECTED</td>
<td>Binding Extensions <a href="https://tools.ietf.org/html/rfc5842">RFC 5842</a>, Section 7.2 (Experimental)</td>
</tr>
<tr>
<td>510</td>
<td>NOT_EXTENDED</td>
<td>An HTTP Extension Framework <a href="https://tools.ietf.org/html/rfc2774">RFC 2774</a>, Section 7 (Experimental)</td>
</tr>
<tr>
<td>511</td>
<td>NETWORK_AUTHENTICATION_REQUIRED</td>
<td>Additional HTTP Status Codes <a href="https://tools.ietf.org/html/rfc6585">RFC 6585</a>, Section 6</td>
</tr>
</tbody>
</table>

In order to preserve backwards compatibility, enum values are also present in the `http.client` module in the form of constants. The enum name is equal to the constant name (i.e. `http.HTTPStatus.OK` is also available as `http.client.OK`).

*Changed in version 3.7*: Added `421 MISDIRECTED_REQUEST` status code.
http.client — HTTP protocol client

Source code: Lib/http/client.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module urllib.request uses it to handle URLs that use HTTP and HTTPS.

See also: The Requests package is recommended for a higher-level HTTP client interface.

Note: HTTPS support is only available if Python was compiled with SSL support (through the ssl module).

The module provides the following classes:

class http.client.HTTPConnection(host, port=None, [timeout, ]source_address=, blocksize=8192)

    An HTTPConnection instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form host:port, else the default HTTP port (80) is used. If the optional timeout parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional source_address parameter may be a tuple of a (host, port) to use as the source
address the HTTP connection is made from. The optional `blocksize` parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```python
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org')
>>> h4 = http.client.HTTPConnection('www.python.org')
```

*Changed in version 3.2:* `source_address` was added.

*Changed in version 3.4:* The `strict` parameter was removed. HTTP 0.9-style “Simple Responses” are not longer supported.

*Changed in version 3.7:* `blocksize` parameter was added.

```python
class http.client.HTTPSConnection(host, port=None, key_file=None, cert_file=None, [timeout, ]source_address=None, *, context=None, check_hostname=None, blocksize=8192)
```

A subclass of `HTTPConnection` that uses SSL for communication with secure servers. Default port is 443. If `context` is specified, it must be a `ssl.SSLContext` instance describing the various SSL options.

Please read Security considerations for more information on best practices.

*Changed in version 3.2:* `source_address`, `context` and `check_hostname` were added.
Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if `ssl.HAS_SNI` is true).

Changed in version 3.4: The `strict` parameter was removed. HTTP 0.9-style “Simple Responses” are no longer supported.

Changed in version 3.4.3: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the `context` parameter.

Deprecated since version 3.6: `key_file` and `cert_file` are deprecated in favor of `context`. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system’s trusted CA certificates for you.

The `check_hostname` parameter is also deprecated; the `ssl.SSLContext.check_hostname` attribute of `context` should be used instead.

class http.client.HTTPResponse(sock, debuglevel=0, method=None, url=None)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

Changed in version 3.4: The `strict` parameter was removed. HTTP 0.9 style “Simple Responses” are no longer supported.

The following exceptions are raised as appropriate:

exception http.client.HTTPException
The base class of the other exceptions in this module. It is a subclass of `Exception`.

`exception` `http.client.NotConnected`  
A subclass of `HTTPException`.

`exception` `http.client.InvalidURL`  
A subclass of `HTTPException`, raised if a port is given and is either non-numeric or empty.

`exception` `http.client.UnknownProtocol`  
A subclass of `HTTPException`.

`exception` `http.client.UnknownTransferEncoding`  
A subclass of `HTTPException`.

`exception` `http.client.UnimplementedFileMode`  
A subclass of `HTTPException`.

`exception` `http.client.IncompleteRead`  
A subclass of `HTTPException`.

`exception` `http.client.ImproperConnectionState`  
A subclass of `HTTPException`.

`exception` `http.client.CannotSendRequest`  
A subclass of `ImproperConnectionState`.

`exception` `http.client.CannotSendHeader`  
A subclass of `ImproperConnectionState`.

`exception` `http.client.ResponseNotReady`  
A subclass of `ImproperConnectionState`. 
exception `http.client.BadStatusLine`

A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don’t understand.

exception `http.client.LineTooLong`

A subclass of `HTTPException`. Raised if an excessively long line is received in the HTTP protocol from the server.

exception `http.client.RemoteDisconnected`

A subclass of `ConnectionResetError` and `BadStatusLine`. Raised by `HTTPConnection.getresponse()` when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

*New in version 3.5:* Previously, `BadStatusLine('')` was raised.

The constants defined in this module are:

`http.client.HTTP_PORT`

The default port for the HTTP protocol (always 80).

`http.client.HTTPS_PORT`

The default port for the HTTPS protocol (always 443).

`http.client.responses`

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example:

```
http.client.responses[http.client.NOT_FOUND]  # is 'Not Found'
```

See [HTTP status codes](https://httpstatuses.com) for a list of HTTP status codes that are
available in this module as constants.
HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.request(method, url, body=None, headers={}, *, encode_chunked=False)

This will send a request to the server using the HTTP request method method and the selector url.

If body is specified, the specified data is sent after the headers are finished. It may be a str, a bytes-like object, an open file object, or an iterable of bytes. If body is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a file object, the contents of the file are sent; this file object should support at least the read() method. If the file object is an instance of io.TextIOBase, the data returned by the read() method will be encoded as ISO-8859-1, otherwise the data returned by read() is sent as is. If body is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The headers argument should be a mapping of extra HTTP headers to send with the request.

If headers contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If body is None, the Content-Length header is set to 0 for methods that expect a body (PUT, POST, and PATCH). If body is a string or a bytes-like object that is not also a file, the Content-Length header is set to its length. Any other type of body (files and iterables in general) will be chunk-encoded, and the
Transfer-Encoding header will automatically be set instead of Content-Length.

The `encode_chunked` argument is only relevant if Transfer-Encoding is specified in `headers`. If `encode_chunked` is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

**Note:** Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a `str` or bytes-like object that is not also a file as the body representation.

*New in version 3.2:* `body` can now be an iterable.

*Changed in version 3.6:* If neither Content-Length nor Transfer-Encoding are set in `headers`, file and iterable `body` objects are now chunk-encoded. The `encode_chunked` argument was added. No attempt is made to determine the Content-Length for file objects.

**HTTPConnection.getresponse()**

Should be called after a request is sent to get the response from the server. Returns an `HTTPResponse` instance.

**Note:** Note that you must have read the whole response before you can send a new request to the server.

*Changed in version 3.5:* If a `ConnectionError` or subclass is raised, the `HTTPConnection` object will be ready to reconnect when a new request is sent.

**HTTPConnection.set_debuglevel(level)**
Set the debugging level. The default debug level is 0, meaning no debugging output is printed. Any value greater than 0 will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new `HTTPResponse` objects that are created.

*New in version 3.1.*

`HTTPConnection.set_tunnel(host, port=None, headers=None)`  
Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, not the address of the proxy server).

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the `HTTPSConnection` constructor, and the address of the host that we eventually want to reach to the `set_tunnel()` method:

```python
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

*New in version 3.2.*

`HTTPConnection.connect()`  
Connect to the server specified when the object was created. By
default, this is called automatically when making a request if the client does not already have a connection.

HTTPConnection.close()
Close the connection to the server.

HTTPConnection.blocksize
Buffer size in bytes for sending a file-like message body.

New in version 3.7.

As an alternative to using the request() method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.putrequest(method, url, skip_host=False, skip_accept_encoding=False)
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the method string, the url string, and the HTTP version (HTTP/1.1). To disable automatic sending of Host: or Accept-Encoding: headers (for example to accept additional content encodings), specify skip_host or skip_accept_encoding with non-False values.

HTTPConnection.putheader(header, argument[, ...])
Send an RFC 822-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.endheaders(message_body=None, *, encode_chunked=False)
Send a blank line to the server, signalling the end of the headers.
The optional message_body argument can be used to pass a message body associated with the request.

If encode_chunked is `True`, the result of each iteration of message_body will be chunk-encoded as specified in RFC 7230, Section 3.3.1. How the data is encoded is dependent on the type of message_body. If message_body implements the buffer interface the encoding will result in a single chunk. If message_body is a collections.abc.Iterable, each iteration of message_body will result in a chunk. If message_body is a file object, each call to .read() will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after message_body.

**Note:** Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

*New in version 3.6:* Chunked encoding support. The encode_chunked parameter was added.

```
HTTPConnection.send(data)
```

Send data to the server. This should be used directly only after the endheaders() method has been called and before getresponse() is called.
HTTPResponse Objects

An `HTTPResponse` instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

*Changed in version 3.5:* The `io.BufferedIOBase` interface is now implemented and all of its reader operations are supported.

`HTTPResponse.read([amt])`

Reads and returns the response body, or up to the next `amt` bytes.

`HTTPResponse.readinto(b)`

Reads up to the next `len(b)` bytes of the response body into the buffer `b`. Returns the number of bytes read.

*New in version 3.3.*

`HTTPResponse.getheader(name, default=None)`

Return the value of the header `name`, or `default` if there is no header matching `name`. If there is more than one header with the name `name`, return all of the values joined by ‘, ‘. If ‘default’ is any iterable other than a single string, its elements are similarly returned joined by commas.

`HTTPResponse.getheaders()`

Return a list of (header, value) tuples.

`HTTPResponse.fileno()`

Return the `fileno` of the underlying socket.
HTTPResponse. **msg**

HTTPResponse. **version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse. **status**
Status code returned by server.

HTTPResponse. **reason**
Reason phrase returned by server.

HTTPResponse. **debuglevel**
A debugging hook. If debuglevel is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse. **closed**
Is True if the stream is closed.
Examples

Here is an example session that uses the GET method:

```python
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
# The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200))  # 200 bytes
b'<!doctype html>\n
''--[if"
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the HEAD method. Note that the HEAD method never returns any data.

```python
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
```
Here is an example session that shows how to POST requests:

```python
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524})
>>> headers = {'Content-type': 'application/x-www-form-urlencoded',
...            'Accept': 'text/plain'}
>>> conn = http.client.HTTPConnection('bugs.python.org')
>>> conn.request('POST', '', params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/issue12524</a>'
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only on the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods are also handled in urllib.request.Request by sending the appropriate method attribute. Here is an example session that shows how to do PUT request using http.client:

```python
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
```
>>> # for the resource http://localhost:8080/file
... import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
HTTPMessage Objects

An http.client.HTTPMessage instance holds the headers from an HTTP response. It is implemented using the email.message.Message class.
ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module urllib.request to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```python
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')   # connect to host,
>>> ftp.login()                  # user anonymous
'230 Login successful.'
>>> ftp.cwd('debian')           # change into "debian" directory
>>> ftp.retrlines('LIST')       # list directory
-rw-rw-r--  1 1176  1176  1063  Jun 15 10:18 README...
...  
drwxr-sr-x  5 1176  1176  4096  Dec 19  2000 pool
drwxr-sr-x  4 1176  1176  4096  Nov 17  2008 project
drwxr-xr-x  3 1176  1176  4096  Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb'))
'226 Transfer complete.'
>>> ftp.quit()
```

The module defines the following items:

```python
class ftplib.FTP(host='', user='', passwd='', acct='', timeout=None,
```
source_address=None)

Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made (where passwd and acct default to the empty string when not given). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). source_address is a 2-tuple (host, port) for the socket to bind to as its source address before connecting.

The FTP class supports the with statement, e.g.:

```python
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
... # doctest: +SKIP
230 Anonymous login ok, restrictions apply.
dr-xr-xr-x 9 ftp ftp 154 May 6
dr-xr-xr-x 9 ftp ftp 154 May 6
dr-xr-xr-x 5 ftp ftp 4096 May 6
dr-xr-xr-x 3 ftp ftp 18 Jul 10
```

Changed in version 3.2: Support for the with statement was added.

Changed in version 3.3: source_address parameter was added.

class ftplib.FTP_TLS(host='', user='', passwd='', acct='',
keyfile=None, certfile=None, context=None, timeout=None,
source_address=None)
A FTP subclass which adds TLS support to FTP as described in RFC 4217. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the `prot_p()` method. `context` is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

`keyfile` and `certfile` are a legacy alternative to `context` – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

New in version 3.2.

Changed in version 3.3: `source_address` parameter was added.

Changed in version 3.4: The class now supports hostname check with `ssl.SSLContext.check_hostname` and Server Name Indication (see `ssl.HAS_SNI`).

Deprecated since version 3.6: `keyfile` and `certfile` are deprecated in favor of `context`. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

Here's a sample session using the FTP_TLS class:

```python
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"
```
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bs...

exception ftplib.error_reply
Exception raised when an unexpected reply is received from the server.

exception ftplib.error_temp
Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.error_perm
Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.error_proto
Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.all_errors
The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as OSError.

See also:

Module netrc
Parser for the .netrc file format. The file .netrc is typically used by FTP clients to load user authentication information before prompting the user.
FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

**FTP.set_debuglevel(level)**

Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

**FTP.connect(host=", port=0, timeout=None, source_address=None)**

Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional `timeout` parameter specifies a timeout in seconds for the connection attempt. If no `timeout` is passed, the global default timeout setting will be used. `source_address` is a 2-tuple (host, port) for the socket to bind to as its source address before connecting.

*Changed in version 3.3: source_address parameter was added.*
FTP. `getwelcome()`
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

FTP. `login(user='anonymous', passwd='', acct='')`
Log in as the given `user`. The `passwd` and `acct` parameters are optional and default to the empty string. If no `user` is specified, it defaults to 'anonymou'. If `user` is 'anonymous', the default `passwd` is 'anonymous@'. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The `acct` parameter supplies “accounting information”; few systems implement this.

FTP. `abort()`
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

FTP. `sendcmd(cmd)`
Send a simple command string to the server and return the response string.

FTP. `voidcmd(cmd)`
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise `error_reply` otherwise.

FTP. `retrbinary(cmd, callback, blocksize=8192, rest=None)`
Retrieve a file in binary transfer mode. `cmd` should be an
appropriate RETR command: 'RETR filename'. The callback function is called for each block of data received, with a single bytes argument giving the data block. The optional blocksize argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to callback). A reasonable default is chosen. rest means the same thing as in the transfercmd() method.

**FTP**: retrlines(cmd, callback=None)

Retrieve a file or directory listing in ASCII transfer mode. cmd should be an appropriate RETR command (see retrbinary()) or a command such as LIST or NLST (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. The callback function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default callback prints the line to sys.stdout.

**FTP**: set_pasv(val)

Enable “passive” mode if val is true, otherwise disable passive mode. Passive mode is on by default.

**FTP**: storbinary(cmd, fp, blocksize=8192, callback=None, rest=None)

Store a file in binary transfer mode. cmd should be an appropriate STOR command: "STOR filename". fp is a file object (opened in binary mode) which is read until EOF using its read() method in blocks of size blocksize to provide the data to be stored. The blocksize argument defaults to 8192. callback is an optional single parameter callable that is called on each block of data after it is sent. rest means the same thing as in the transfercmd()
method.

*Changed in version 3.2: rest parameter added.*

FTP. **storlines**(cmd, fp, callback=None)

Store a file in ASCII transfer mode. `cmd` should be an appropriate STOR command (see `storbinary()`). Lines are read until EOF from the file object `fp` (opened in binary mode) using its `readline()` method to provide the data to be stored. `callback` is an optional single parameter callable that is called on each line after it is sent.

FTP. **transfercmd**(cmd, rest=None)

Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by `cmd`, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional `rest` is given, a REST command is sent to the server, passing `rest` as an argument. `rest` is usually a byte offset into the requested file, telling the server to restart sending the file’s bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that `rest` be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts `rest` to a string, but no check is performed on the string’s contents. If the server does not recognize the REST command, an error_reply exception will be raised. If this happens, simply call `transfercmd()` without a `rest` argument.
FTP. ntransfercmd(cmd, rest=None)
Like transfercmd(), but returns a tuple of the data connection
and the expected size of the data. If the expected size could not be
computed, None will be returned as the expected size. cmd and
rest means the same thing as in transfercmd().
FTP. mlsd(path="", facts=[])
List a directory in a standardized format by using MLSD command
(RFC 3659). If path is omitted the current directory is assumed.
facts is a list of strings representing the type of information desired
(e.g. ["type", "size", "perm"]). Return a generator object
yielding a tuple of two elements for every file found in path. First
element is the file name, the second one is a dictionary containing
facts about the file name. Content of this dictionary might be limited
by the facts argument but server is not guaranteed to return all
requested facts.
New in version 3.3.
FTP. nlst(argument[, ...])
Return a list of file names as returned by the NLST command. The
optional argument is a directory to list (default is the current server
directory). Multiple arguments can be used to pass non-standard
options to the NLST command.
Note: If your server supports the command, mlsd() offers a
better API.
FTP. dir(argument[, ...])
Produce a directory listing as returned by the LIST command,
printing it to standard output. The optional argument is a directory


to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the LIST command. If the last argument is a function, it is used as a callback function as for retrlines(); the default prints to sys.stdout. This method returns None.

**Note:** If your server supports the command, mlsd() offers a better API.

FTP. **rename**(fromname, toname)
Rename file fromname on the server to toname.

FTP. **delete**(filename)
Remove the file named filename from the server. If successful, returns the text of the response, otherwise raises error_perm on permission errors or error_reply on other errors.

FTP. **cwd**(pathname)
Set the current directory on the server.

FTP. **mkd**(pathname)
Create a new directory on the server.

FTP. **pwd**()
Return the pathname of the current directory on the server.

FTP. **rmd**(dirname)
Remove the directory named dirname on the server.

FTP. **size**(filename)
Request the size of the file named filename on the server. On success, the size of the file is returned as an integer, otherwise
None is returned. Note that the SIZE command is not standardized, but is supported by many common server implementations.

**FTP.** `quit()`

Send a QUIT command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception if the server responds with an error to the QUIT command. This implies a call to the `close()` method which renders the FTP instance useless for subsequent calls (see below).

**FTP.** `close()`

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the FTP instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).
FTP_TLS Objects

**FTP_TLS** class inherits from **FTP**, defining these additional objects:

**FTP_TLS.** **ssl_version**

The SSL version to use (defaults to **ssl.PROTOCOL_SSLv23**).

**FTP_TLS.** **auth()**

Set up a secure control connection by using TLS or SSL, depending on what is specified in the **ssl_version** attribute.

*Changed in version 3.4:* The method now supports hostname check with **ssl.SSLContext.check_hostname** and **Server Name Indication** (see **ssl.HAS_SNI**).

**FTP_TLS.** **ccc()**

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

*New in version 3.3.*

**FTP_TLS.** **prot_p()**

Set up secure data connection.

**FTP_TLS.** **prot_c()**

Set up clear text data connection.
This module defines a class, **POP3**, which encapsulates a connection to a POP3 server and implements the protocol as defined in [RFC 1939](https://tools.ietf.org/html/rfc1939). The **POP3** class supports both the minimal and optional command sets from [RFC 1939](https://tools.ietf.org/html/rfc1939). The **POP3** class also supports the STLS command introduced in [RFC 2595](https://tools.ietf.org/html/rfc2595) to enable encrypted communication on an already established connection.

Additionally, this module provides a class **POP3_SSL**, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the `imaplib.IMAP4` class, as IMAP servers tend to be better implemented.

The **poplib** module provides two classes:

```python
class poplib.POP3(host, port=POP3_PORT[, timeout])
```

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If `port` is omitted, the standard POP3 port (110) is used. The optional `timeout` parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).
class poplib.**POP3_SSL**(host, port=POP3_SSL_PORT, keyfile=None, certfile=None, timeout=None, context=None)

This is a subclass of **POP3** that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the **POP3** constructor. *context* is an optional `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

*keyfile* and *certfile* are a legacy alternative to *context* - they can point to PEM-formatted private key and certificate chain files, respectively, for the SSL connection.

*Changed in version 3.2:* *context* parameter added.

*Changed in version 3.4:* The class now supports hostname check with `ssl.SSLContext.check_hostname` and Server Name Indication (see `ssl.HAS_SNI`).

*Deprecated since version 3.6:* *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system’s trusted CA certificates for you.

One exception is defined as an attribute of the `poplib` module:

*exception* poplib.**error_proto**

Exception raised on any errors from this module (errors from `socket` module are not caught). The reason for the exception is passed to the constructor as a string.
See also:

**Module imaplib**
The standard Python IMAP module.

**Frequently Asked Questions About Fetchmail**
The FAQ for the fetchmail POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.
# POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An **POP3** instance has the following methods:

**POP3**. `set_debuglevel(level)`
- Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

**POP3**. `getwelcome()`
- Returns the greeting string sent by the POP3 server.

**POP3**. `capa()`
- Query the server's capabilities as specified in RFC 2449. Returns a dictionary in the form `{ 'name': ['param'[...]] }`.

*New in version 3.4.*

**POP3**. `user(username)`
- Send user command, response should indicate that a password is required.

**POP3**. `pass_(password)`
- Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.
**POP3. apop***(user, secret)**

Use the more secure APOP authentication to log into the POP3 server.

**POP3. rpop**(user)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

**POP3. stat()**

Get mailbox status. The result is a tuple of 2 integers: *(message count, mailbox size)*.

**POP3. list([which])**

Request message list, result is in the form *(response, ['mesg_num octets', ...], octets)*. If *which* is set, it is the message to list.

**POP3. retr(which)**

Retrieve whole message number *which*, and set its seen flag. Result is in form *(response, ['line', ...], octets)*.

**POP3. dele(which)**

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

**POP3. rset()**

Remove any deletion marks for the mailbox.

**POP3. noop()**

Do nothing. Might be used as a keep-alive.
POP3. **quit**()
Signoff: commit changes, unlock mailbox, drop connection.

POP3. **top**(which, howmuch)
Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form (response, ['line', ...], octets).

The POP3 TOP command this method uses, unlike the RETR command, doesn’t set the message’s seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3. **uidl**(which=None)
Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form 'response mesgnum uid', otherwise result is list (response, ['mesgnum uid', ...], octets).

POP3. **utf8**()
Try to switch to UTF-8 mode. Returns the server response if successful, raises *error_proto* if not. Specified in RFC 6856.

*New in version 3.5.*

POP3. **stls**(context=None)
Start a TLS session on the active connection as specified in RFC 2595. This is only allowed before user authentication.

*context* parameter is a ssl.SSLContext object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security
considerations for best practices.

This method supports hostname checking via
ssl.SSLContext.check_hostname and Server Name
Indication (see ssl.HAS_SNI).

New in version 3.4.

Instances of POP3_SSL have no additional methods. The interface of
this subclass is identical to its parent.
POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.
**imaplib — IMAP4 protocol client**

**Source code:** Lib/imaplib.py

This module defines three classes, `IMAP4`, `IMAP4_SSL` and `IMAP4_stream`, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in RFC 2060. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the `STATUS` command is not supported in IMAP4.

Three classes are provided by the `imaplib` module, `IMAP4` is the base class:

```python
class imaplib.IMAP4(host='', port=IMAP4_PORT)
```

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If `host` is not specified, '' (the local host) is used. If `port` is omitted, the standard IMAP4 port (143) is used.

The `IMAP4` class supports the `with` statement. When used like this, the IMAP4 LOGOUT command is issued automatically when the `with` statement exits. E.g.:

```python
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25f65hy903weo.8
```
Changed in version 3.5: Support for the `with` statement was added.

Three exceptions are defined as attributes of the `IMAP4` class:

**exception IMAP4.error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

**exception IMAP4.abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of `IMAP4.error`. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

**exception IMAP4.readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of `IMAP4.error`. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

```python
class imaplib.IMAP4_SSL(host='', port=IMAP4_SSL_PORT,
keyfile=None, certfile=None, ssl_context=None)
```

This is a subclass derived from `IMAP4` that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If `host` is not specified, `'.'` (the local host) is used. If `port` is omitted, the standard IMAP4-over-SSL port (993) is used. `ssl_context` is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read `Security considerations` for best practices.
*keyfile* and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile/certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile/certfile* is provided along with *ssl_context*.

*Changed in version 3.3:* *ssl_context* parameter added.

*Changed in version 3.4:* The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

*Deprecated since version 3.6:* *keyfile* and *certfile* are deprecated in favor of *ssl_context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system’s trusted CA certificates for you.

The second subclass allows for connections created by a child process:

```python
class imaplib.IMAP4_stream(command)
```

This is a subclass derived from *IMAP4* that connects to the stdin/stdout file descriptors created by passing *command* to *subprocess.Popen()*.

The following utility functions are defined:

```python
imaplib/Internaldate2tuple(datestr)
```

Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a *time.struct_time* tuple or None if the string has wrong format.
imaplib.**Int2AP**(num)
Converts an integer into a string representation using characters from the set [A .. P].

imaplib.**ParseFlags**(flagstr)
Converts an IMAP4 FLAGS response to a tuple of individual flags.

imaplib.**Time2Internaldate**(date_time)
Converts *date_time* to an IMAP4 INTERNALDATE representation. The return value is a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by `time.time()`), a 9-tuple representing local time an instance of `time.struct_time` (as returned by `time.localtime()`), an aware instance of `datetime.datetime`, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an EXPUNGE command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

**See also:** Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington’s IMAP Information Center ([https://www.washington.edu/imap/](https://www.washington.edu/imap/)).
IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn’t enclosed with either parentheses or double quotes) each string is quoted. However, the password argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the flags argument to STORE) then enclose the string in parentheses (eg: `r'\Deleted'`).

Each command returns a tuple: `(type, [data, ...])` where `type` is usually `'OK'` or `'NO'`, and `data` is either the text from the command response, or mandated results from the command. Each `data` is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: ‘literal’ value).

The `message_set` options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An IMAP4 instance has the following methods:

```
IMAP4.append(mailbox, flags, date_time, message)
```

Append `message` to named mailbox.
IMAP4. authenticate(mechanism, authobject)

Authenticate command — requires response processing.

*mechanism* specifies which authentication mechanism is to be used - it should appear in the instance variable *capabilities* in the form AUTH=mechanism.

*authobject* must be a callable object:

```python
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be bytes. It should return bytes *data* that will be base64 encoded and sent to the server. It should return *None* if the client abort response ⌂ should be sent instead.

*Changed in version 3.5:* string usernames and passwords are now encoded to utf-8 instead of being limited to ASCII.

IMAP4. check()

Checkpoint mailbox on server.

IMAP4. close()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before LOGOUT.

IMAP4. copy(message_set, new_mailbox)

Copy *message_set* messages onto end of *new_mailbox*.

IMAP4. create(mailbox)

Create new mailbox named *mailbox*. 
**IMAP4. `delete(mailbox)`**
Delete old mailbox named `mailbox`.

**IMAP4. `deleteacl(mailbox, who)`**
Delete the ACLs (remove any rights) set for `who` on `mailbox`.

**IMAP4. `enable(capability)`**
Enable `capability` (see RFC 5161). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see RFC 6855).

*New in version 3.5:* The `enable()` method itself, and RFC 6855 support.

**IMAP4. `expunge()`**
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

**IMAP4. `fetch(message_set, message_parts)`**
Fetch (parts of) messages. `message_parts` should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

**IMAP4. `getacl(mailbox)`**
Get the ACLs for `mailbox`. The method is non-standard, but is supported by the Cyrus server.

**IMAP4. `getannotation(mailbox, entry, attribute)`**
Retrieve the specified ANNOTATIONs for `mailbox`. The method is
non-standard, but is supported by the Cyrus server.

**IMAP4. `getquota(root)`**
Get the quota root’s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

**IMAP4. `getquotaroot(mailbox)`**
Get the list of quota roots for the named mailbox. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

**IMAP4. `list([directory[, pattern]])`**
List mailbox names in directory matching pattern. directory defaults to the top-level mail folder, and pattern defaults to match anything. Returned data contains a list of LIST responses.

**IMAP4. `login(user, password)`**
Identify the client using a plaintext password. The password will be quoted.

**IMAP4. `login_cram_md5(user, password)`**
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

**IMAP4. `logout()`**
Shutdown connection to server. Returns server BYE response.

**IMAP4. `lsub(directory='', pattern='*')`**
List subscribed mailbox names in directory matching pattern. directory defaults to the top level directory and pattern defaults to match any mailbox. Returned data are tuples of message part envelope and data.
**IMAP4. `myrights(mailbox)`**
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

**IMAP4. `namespace()`**
Returns IMAP namespaces as defined in [RFC 2342](https://tools.ietf.org/html/rfc2342).

**IMAP4. `noop()`**
Send `NOOP` to server.

**IMAP4. `open(host, port)`**
Opens socket to `port` at `host`. This method is implicitly called by the `IMAP4` constructor. The connection objects established by this method will be used in the `IMAP4.read()`, `IMAP4.readline()`, `IMAP4.send()`, and `IMAP4.shutdown()` methods. You may override this method.

**IMAP4. `partial(message_num, message_part, start, length)`**
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

**IMAP4. `proxyauth(user)`**
Assume authentication as `user`. Allows an authorised administrator to proxy into any user's mailbox.

**IMAP4. `read(size)`**
Reads `size` bytes from the remote server. You may override this method.

**IMAP4. `readline()`**
Reads one line from the remote server. You may override this method.
**IMAP4. recent()**

Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

**IMAP4. rename(oldmailbox, newmailbox)**

Rename mailbox named `oldmailbox` to `newmailbox`.

**IMAP4. response(code)**

Return data for response `code` if received, or `None`. Returns the given code, instead of the usual type.

**IMAP4. search(charset, criterion[, ...])**

Search mailbox for matching messages. `charset` may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. `charset` must be `None` if the `UTF8=ACCEPT` capability was enabled using the `enable()` command.

Example:

```python
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')
# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

**IMAP4. select(mailbox='INBOX', readonly=False)**

Select a mailbox. Returned data is the count of messages in `mailbox` (EXISTS response). The default `mailbox` is `'INBOX'`. If the `readonly` flag is set, modifications to the mailbox are not allowed.
**IMAP4.** `send(data)`
Sends data to the remote server. You may override this method.

**IMAP4.** `setacl(mailbox, who, what)`
Set an ACL for mailbox. The method is non-standard, but is supported by the Cyrus server.

**IMAP4.** `setannotation(mailbox, entry, attribute[,...])`
Set ANNOTATIONs for mailbox. The method is non-standard, but is supported by the Cyrus server.

**IMAP4.** `setquota(root, limits)`
Set the quota root’s resource limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

**IMAP4.** `shutdown()`
Close connection established in open. This method is implicitly called by `IMAP4.logout()`. You may override this method.

**IMAP4.** `socket()`
Returns socket instance used to connect to server.

**IMAP4.** `sort(sort_criteria, charset, search_criterion[,...])`
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the `search_criterion` argument(s); a parenthesized list of `sort_criteria`, and the searching `charset`. Note that unlike `search`, the searching `charset` argument is mandatory. There is also a `uid` sort command which corresponds to sort the way that `uid` search corresponds to search. The sort
command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

**IMAP4.** starttls(ssl_context=None)
Send a STARTTLS command. The *ssl_context* argument is optional and should be a ssl.SSLContext object. This will enable encryption on the IMAP connection. Please read Security considerations for best practices.

*New in version 3.2.*

*Changed in version 3.4:* The method now supports hostname check with ssl.SSLContext.check_hostname and Server Name Indication (see ssl.HAS_SNI).

**IMAP4.** status(mailbox, names)
Request named status conditions for *mailbox*.

**IMAP4.** store(message_set, command, flag_list)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of RFC 2060 as being one of “FLAGS”, “+FLAGS”, or “-FLAGS”, optionally with a suffix of “.SILENT”.

For example, to set the delete flag on all messages:

```python
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\Deleted')
M.expunge()
```
Note: Creating flags containing ‘]’ (for example: “[test]”) violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

**IMAP4. `subscribe`**(mailbox)
Subscribe to new mailbox.

**IMAP4. `thread`**(threading_algorithm, charset, search_criterion[, ...])
The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the `search_criterion` argument(s); a `threading_algorithm`, and the searching `charset`. Note that unlike `search`, the searching `charset` argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.
This is an IMAP4rev1 extension command.

**IMAP4.** `uid(command, arg[, ...])`
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

**IMAP4.** `unsubscribe(mailbox)`
Unsubscribe from old mailbox.

**IMAP4.** `xatom(name[, ...])`
Allow simple extension commands notified by server in CAPABILITY response.

The following attributes are defined on instances of **IMAP4**:

**IMAP4.** `PROTOCOL_VERSION`
The most recent supported protocol in the CAPABILITY response from the server.

**IMAP4.** `debug`
Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

**IMAP4.** `utf8_enabled`
Boolean value that is normally `False`, but is set to `True` if an `enable()` command is successfully issued for the `UTF8=ACCEPT` capability.

*New in version 3.5.*
IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
ty, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s
%s
% (num, data[0][1]))
M.close()
M.logout()
```
nntplib — NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class NNTP which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with RFC 3977 as well as the older RFC 977 and RFC 2980.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```python
>>> s = nntplib.NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...    print(id, nntplib.decode_header(over['subject']))
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```
To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following classes:

```
class nntplib.NNTP(host, port=119, user=None, password=None,
readermode=None, usenetrc=False[, timeout])
```

Return a new **NNTP** object, representing a connection to the NNTP server running on host **host**, listening at port **port**. An optional **timeout** can be specified for the socket connection. If the optional **user** and **password** are provided, or if suitable credentials are present in `/.netrc` and the optional flag **usenetrc** is true, the **AUTHINFO USER** and **AUTHINFO PASS** commands are used to identify and authenticate the user to the server. If the optional flag **readermode** is true, then a **mode reader** command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as **group**. If you get unexpected **NNTPPermanentError**s, you might need to set **readermode**. The **NNTP** class supports the **with** statement to unconditionally consume **OSError** exceptions and to close the NNTP connection when done, e.g.:
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...   n.group('gmane.comp.python.committers')
...   # doctest: +SKIP
('211 1755 1 1755 gmane.comp.python.committers',

Changed in version 3.2: usenetrc is now False by default.

Changed in version 3.3: Support for the with statement was added.

class nntplib.NNTP_SSL(host, port=563, user=None, password=None, ssl_context=None, readermode=None, usenetrc=False[, timeout])

Return a new NNTP_SSL object, representing an encrypted connection to the NNTP server running on host host, listening at port port. NNTP_SSL objects have the same methods as NNTP objects. If port is omitted, port 563 (NNTPS) is used. ssl_context is also optional, and is a SSLContext object. Please read Security considerations for best practices. All other parameters behave the same as for NNTP.

Note that SSL-on-563 is discouraged per RFC 4642, in favor of STARTTLS as described below. However, some servers only support the former.

New in version 3.2.

Changed in version 3.4: The class now supports hostname check with ssl.SSLContext.check_hostname and Server Name Indication (see ssl.HAS_SNI).
exception nntplib.NNTPError
   Derived from the standard exception Exception, this is the base class for all exceptions raised by the nntplib module. Instances of this class have the following attribute:

   **response**
   The response of the server if available, as a str object.

exception nntplib.NNTPRetryError
   Exception raised when an unexpected reply is received from the server.

exception nntplib.NNTPTemporaryError
   Exception raised when a response code in the range 400–499 is received.

exception nntplib.NNTPPermanentError
   Exception raised when a response code in the range 500–599 is received.

exception nntplib.NNTPProtocolError
   Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception nntplib.NNTpDataError
   Exception raised when there is some error in the response data.
NNTP Objects

When connected, NNTP and NNTP_SSL objects support the following methods and attributes.

Attributes

**NNTP.** **nntp_version**

An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising **RFC 3977** compliance and 1 for others.

*New in version 3.2.*

**NNTP.** **nntp_implementation**

A string describing the software name and version of the NNTP server, or **None** if not advertised by the server.

*New in version 3.2.*

Methods

The *response* that is returned as the first item in the return tuple of almost all methods is the server’s response: a string beginning with a three-digit code. If the server’s response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument **file**. When the **file** argument is supplied, it must be either a file object opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the
server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

*Changed in version 3.2:* Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

**NNTP.** **quit()**

Send a `QUIT` command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

**NNTP.** **getwelcome()**

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

**NNTP.** **getcapabilities()**

Return the RFC 3977 capabilities advertised by the server, as a `dict` instance mapping capability names to (possibly empty) lists of values. On legacy servers which don’t understand the `CAPABILITIES` command, an empty dictionary is returned instead.

```python
>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True
```

*New in version 3.2.*

**NNTP.** **login**(user=None, password=None, usenetrc=True)

Send `AUTHINFO` commands with the user name and password. If
user and password are None and usenetrc is true, credentials from ~/.netrc will be used if possible.

Unless intentionally delayed, login is normally performed during the NNTP object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set user or password when creating the object, and must set usenetrc to False.

New in version 3.2.

NNTP.starttls(context=None)
Send a STARTTLS command. This will enable encryption on the NNTP connection. The context argument is optional and should be a ssl.SSLContext object. Please read Security considerations for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a NNTP object initialization. See NNTP.login() for information on suppressing this behavior.

New in version 3.2.

Changed in version 3.4: The method now supports hostname check with ssl.SSLContext.check_hostname and Server Name Indication (see ssl.HAS_SNI).

NNTP.newgroups(date, *, file=None)
Send a NEWGROUPS command. The date argument should be a datetime.date or datetime.datetime object. Return a pair (response, groups) where groups is a list representing the groups that are new since the given date. If file is supplied, though,
then groups will be empty.

```python
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today()) - timedelta
>>> len(groups) # doctest: +SKIP
85
>>> groups[0] # doctest: +SKIP
GroupInfo(group='gmane.network.tor.devel', last='1',
```

NNTP. **newnews** *(group, date, *, file=None)*

Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* has the same meaning as for **newgroups()**. Return a pair *(response, articles)* where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP. **list** *(group_pattern=None, *, file=None)*

Send a LIST or LIST ACTIVE command. Return a pair *(response, list)* where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form *(group, last, first, flag)*, where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- **y**: Local postings and articles from peers are allowed.
- **m**: The group is moderated and all postings must be approved.
- **n**: No local postings are allowed, only articles from peers.
- **j**: Articles from peers are filed in the junk group instead.
- **x**: No local postings, and articles from peers are ignored.
* =foo.bar*: Articles are filed in the foo.bar group instead.

If flag has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if group_pattern is not specified. It is best to cache the results offline unless you really need to refresh them.

*Changed in version 3.2: group_pattern was added.*

**NNTP.** descriptions**(grouppattern)**

Send a LIST NEWSGROUPS command, where grouppattern is a wildmat string as specified in RFC 3977 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, descriptions), where descriptions is a dictionary mapping group names to textual descriptions.

```python
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)  # doctest: +SKIP
295
```

**NNTP.** description**(group)**

Get a description for a single group group. If more than one group matches (if ‘group’ is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use descriptions().

**NNTP.** group**(name)**
Send a **GROUP** command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple *(response, count, first, last, name)* where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

**NNTP**. **over**(*message_spec*, *, file=None*)

Send an **OVER** command, or an **XOVER** command on legacy servers. *message_spec* can be either a string representing a message id, or a *(first, last)* tuple of numbers indicating a range of articles in the current group, or a *(first, None)* tuple indicating a range of articles starting from *first* to the last article in the current group, or **None** to select the current article in the current group.

Return a pair *(response, overviews)*. *overviews* is a list of *(article_number, overview)* tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with "::"). The following items are guaranteed to be present by the NNTP specification:

- the **subject**, **from**, **date**, **message-id** and **references** headers
- the **:bytes** metadata: the number of bytes in the entire raw article (including headers and body)
- the **:lines** metadata: the number of lines in the article body

The value of each item is either a string, or **None** if not present.
It is advisable to use the `decode_header()` function on header values when they may contain non-ASCII characters:

```python
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references'
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>
>>> nntplib.decode_header(over['from'])
'Martin v. Löwis' <martin@v.loewis.de>
```

New in version 3.2.

**NNTP.** `help(*, file=None)`

Send a `HELP` command. Return a pair `(response, list)` where `list` is a list of help strings.

**NNTP.** `stat(message_spec=None)`

Send a `STAT` command, where `message_spec` is either a message id (enclosed in `<' and '>') or an article number in the current group. If `message_spec` is omitted or `None`, the current article in the current group is considered. Return a triple `(response, number, id)` where `number` is the article number and `id` is the message id.

```python
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>
```
NNTP. **next()**

Send a `NEXT` command. Return as for `stat()`.

NNTP. **last()**

Send a `LAST` command. Return as for `stat()`.

NNTP. **article**(message_spec=None, *, file=None)

Send an `ARTICLE` command, where `message_spec` has the same meaning as for `stat()`. Return a tuple `(response, info)` where `info` is a `namedtuple` with three attributes `number`, `message_id` and `lines` (in that order). `number` is the article number in the group (or 0 if the information is not available), `message_id` the message id as a string, and `lines` a list of lines (without terminating newlines) comprising the raw message including headers and body.

```python
>>> resp, info = s.article('20030112190404.GE29873@epoch.metaslash.com')
>>> info.number
0
>>> info.message_id
'20030112190404.GE29873@epoch.metaslash.com'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
'b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b''
```

NNTP. **head**(message_spec=None, *, file=None)

Same as `article()`, but sends a `HEAD` command. The `lines` returned (or written to `file`) will only contain the message headers,
not the body.

NNTP. **body**(message_spec=None, *, file=None)

Same as **article()**, but sends a BODY command. The **lines** returned (or written to **file**) will only contain the message body, not the headers.

NNTP. **post**(data)

Post an article using the POST command. The **data** argument is either a **file object** opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The **post()** method automatically escapes lines beginning with `.` and appends the termination line.

If the method succeeds, the server’s response is returned. If the server refuses posting, a **NNTPReplyError** is raised.

NNTP. **ihave**(message_id, data)

Send an **IHAVE** command. **message_id** is the id of the message to send to the server (enclosed in '<' and '>'). The **data** parameter and the return value are the same as for **post()**.

NNTP. **date**()

Return a pair (response, date). **date** is a **datetime** object containing the current date and time of the server.

NNTP. **slave**()

Send a **SLAVE** command. Return the server's response.

NNTP. **set_debuglevel**(level)

Set the instance’s debugging level. This controls the amount of
debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in RFC 2980. Some of them have been superseded by newer commands in RFC 3977.

**NNTP. xhdr** *(hdr, str, *, file=None)*

Send an XHDR command. The *hdr* argument is a header keyword, e.g. 'subject'. The *str* argument should have the form 'first-last' where *first* and *last* are the first and last article numbers to search. Return a pair *(response, list)*, where *list* is a list of pairs *(id, text)*, where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the XHDR command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a file object, then it will start calling *write()* on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

**NNTP. xover** *(start, end, *, file=None)*

Send an XOVER command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for *over()* . It is recommended to use *over()* instead, since it will automatically use the newer OVER command if available.

**NNTP. xpath** *(id)*


Return a pair \((\text{resp}, \text{path})\), where \(\text{path}\) is the directory path to the article with message ID \(\text{id}\). Most of the time, this extension is not enabled by NNTP server administrators.

*Deprecated since version 3.3:* The XPATH extension is not actively used.
Utility functions

The module also defines the following utility function:

```
nntplib.decode_header(header_str)
```

Decode a header value, un-escaping any escaped non-ASCII characters. `header_str` must be a `str` object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```python
>>> decode_header("Some subject")
'Some subject'
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```
The smtplib module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

```python
class smtplib.SMTP(host='', port=0, local_hostname=None, timeout=None, source_address=None)
```

An SMTP instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP connect() method is called with those parameters during initialization. If specified, local_hostname is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using socket.getfqdn(). If the connect() call returns anything other than a success code, an SMTPConnectError is raised. The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, socket.timeout is raised. The optional source_address parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple (host, port), for the socket to bind to as its source address before connecting. If omitted (or if host or port are '') and/or 0
respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, sendmail(), and SMTP.quit() methods. An example is included below.

The SMTP class supports the with statement. When used like this, the SMTP QUIT command is issued automatically when the with statement exits. E.g.:

```python
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
... (250, b'Ok')
```

*Changed in version 3.3:* Support for the with statement was added.

*Changed in version 3.3:* source_address argument was added.

*New in version 3.5:* The SMTPUTF8 extension ([RFC 6531](https://tools.ietf.org/html/rfc6531)) is now supported.

```python
class smtplib.SMTP_SSL(host='', port=0, local_hostname=None,
keyfile=None, certfile=None, [timeout, ]context=None,
source_address=None)
```

An SMTP_SSL instance behaves exactly the same as instances of SMTP. SMTP_SSL should be used for situations where SSL is required from the beginning of the connection and using starttls() is not appropriate. If host is not specified, the local host is used. If port is zero, the standard SMTP-over-SSL port
(465) is used. The optional arguments `local_hostname`, `timeout` and `source_address` have the same meaning as they do in the SMTP class. `context`, also optional, can contain a `ssl.SSLContext` and allows configuring various aspects of the secure connection. Please read Security considerations for best practices.

`keyfile` and `certfile` are a legacy alternative to `context`, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

*Changed in version 3.3:* `context` was added.

*Changed in version 3.3:* `source_address` argument was added.

*Changed in version 3.4:* The class now supports hostname check with `ssl.SSLContext.check_hostname` and Server Name Indication (see `ssl.HAS_SNI`).

*Deprecated since version 3.6:* `keyfile` and `certfile` are deprecated in favor of `context`. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system’s trusted CA certificates for you.

```python
class smtplib.LMTP(host='', port=LMTP_PORT,
local_hostname=None, source_address=None)
```

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It’s common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments `local_hostname` and `source_address` have the same meaning as they do in the SMTP class. To specify a Unix socket, you must use an absolute path for `host`, starting with a ‘/’. 
Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don’t support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

*exception* `smtplib.SMTPException`

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

*Changed in version 3.4:* `SMTPException` became subclass of `OSError`

*exception* `smtplib.SMTPServerDisconnected`

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the `SMTP` instance before connecting it to a server.

*exception* `smtplib.SMTPResponseException`

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

*exception* `smtplib.SMTPSenderRefused`

Sender address refused. In addition to the attributes set by on all `SMTPResponseException` exceptions, this sets ‘sender’ to the string that the SMTP server refused.

*exception* `smtplib.SMTPRecipientsRefused`

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary
of exactly the same sort as `SMTP.sendmail()` returns.

```python
exception smtplib.SMTPDataError
    The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError
    Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError
    The server refused our HELO message.

exception smtplib.SMTPNotSupportedError
    The command or option attempted is not supported by the server.

    New in version 3.5.

exception smtplib.SMTPAuthenticationError
    SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.
```

See also:

- **RFC 821** - Simple Mail Transfer Protocol
  Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

- **RFC 1869** - SMTP Service Extensions
  Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.
SMTP Objects

An SMTP instance has the following methods:

SMTP. **set_debuglevel**(level)

Set the debug output level. A value of 1 or True for level results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for level results in these messages being timestamped.

*Changed in version 3.5: Added debuglevel 2.*

SMTP. **docmd**(cmd, args="")

Send a command cmd to the server. The optional argument args is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, **SMTPServerDisconnected** will be raised.

SMTP. **connect**(host='localhost', port=0)

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':') followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This
method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP. **helo**(name="")

Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the **helo**_resp attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the **sendmail()** when necessary.

SMTP. **ehlo**(name="")

Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by **has_extn()**. Also sets several informational attributes: the message returned by the server is stored as the **ehlo**_resp attribute, **does_esmtp** is set to true or false depending on whether the server supports ESMTP, and **esmtp_features** will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use **has_extn()** before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by **sendmail()** when necessary.

SMTP. **ehlo_or_helo_if_needed()**

This method calls **ehlo()** and/or **helo()** if there has been no previous **EHLO** or **HELO** command this session. It tries ESMTP
EHLO first.

**SMTPHeloError**

The server didn’t reply properly to the HELO greeting.

**SMTPве́rify**(*address*)

Check the validity of an address on this server using SMTP verify. Returns a tuple consisting of code 250 and a full RFC 822 address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

**Note**: Many sites disable SMTP VRFY in order to foil spammers.

**SMTP.** login(*user*, *password*, *, initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHL0 or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

**SMTPHeloError**

The server didn’t reply properly to the HELO greeting.

**SMTPAuthenticationError**

The server didn’t accept the username/password combination.

**SMTPNotSupportedError**

The AUTH command is not supported by the server.
SMTPException

No suitable authentication method was found.

Each of the authentication methods supported by smtplib are tried in turn if they are advertised as supported by the server. See auth() for a list of supported authentication methods. initial_response_ok is passed through to auth().

Optional keyword argument initial_response_ok specifies whether, for authentication methods that support it, an “initial response” as specified in RFC 4954 can be sent along with the AUTH command, rather than requiring a challenge/response.

*Changed in version 3.5:* SMTPNotSupportedError may be raised, and the initial_response_ok parameter was added.

SMTP.auth(mechanism, authobject, *, initial_response_ok=True)

Issue an SMTP AUTH command for the specified authentication mechanism, and handle the challenge response via authobject.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

\[
\text{data = authobject(challenge=\text{None})}
\]

If optional keyword argument initial_response_ok is true, authobject() will be called first with no argument. It can return the RFC 4954 “initial response” ASCII str which will be encoded and sent with the AUTH command as below. If the authobject()
does not support an initial response (e.g. because it requires a challenge), it should return `None` when called with `challenge=None`. If `initial_response_ok` is false, then `authobject()` will not be called first with `None`.

If the initial response check returns `None`, or if `initial_response_ok` is false, `authobject()` will be called to process the server's challenge response; the `challenge` argument it is passed will be a bytes. It should return ASCII str `data` that will be base64 encoded and sent to the server.

The SMTP class provides authobjects for the CRAM-MD5, PLAIN, and LOGIN mechanisms; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the user and password properties of the SMTP instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the `login()` method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by `smtplib`.

*New in version 3.5.*

**SMTP.starttls**(keyfile=None, certfile=None, context=None)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call `ehlo()` again.

If `keyfile` and `certfile` are provided, they are used to create an
ssl.SSLContext.

Optional *context* parameter is an `ssl.SSLContext` object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

*Deprecated since version 3.6:* *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

**SMTPHeloError**

*The server didn’t reply properly to the `HELO` greeting.*

**SMTPNotSupportedError**

*The server does not support the STARTTLS extension.*

**RuntimeError**

*SSL/TLS support is not available to your Python interpreter.*

*Changed in version 3.3:* *context* was added.

*Changed in version 3.4:* The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see `HAS_SNI`).

*Changed in version 3.5:* The error raised for lack of STARTTLS support is now the `SMTPNotSupportedError` subclass instead of the base `SMTPException`.

`SMTP.sendmail`(*from_addr*, *to_addrs*, *msg*, *mail_options=*()),
rcpt_options=()

Send mail. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as 8bitmime) to be used in MAIL FROM commands as mail_options. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as rcpt_options. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail(), rcpt() and data() to send the message.)

Note: The from_addr and to_addrs parameters are used to construct the message envelope used by the transport agents. sendmail does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone \r and \n characters are converted to \r\n characters. A byte string is not modified.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each
entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If SMTPUTF8 is included in mail_options, and the server supports it, from_addr and to_addrs may contain non-ASCII characters.

This method may raise the following exceptions:

**SMTPRecipientsRefused**
All recipients were refused. Nobody got the mail. The recipients attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

**SMTPHeloError**
The server didn’t reply properly to the HELO greeting.

**SMTPSenderRefused**
The server didn’t accept the from_addr.

**SMTPDataError**
The server replied with an unexpected error code (other than a refusal of a recipient).

**SMTPNotSupportedError**
SMTPUTF8 was given in the mail_options but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

*Changed in version 3.2:* msg may be a byte string.

*Changed in version 3.5:* SMTPUTF8 support added, and SMTPNotSupportedError may be raised if SMTPUTF8 is specified but the server does not support it.
SMTP. **send_message**(msg, from_addr=None, to_addrs=None, 
mail_options=(), rcpt_options=())

This is a convenience method for calling `sendmail()` with the message represented by an `email.message.Message` object. The arguments have the same meaning as for `sendmail()`, except that `msg` is a `Message` object.

If `from_addr` is `None` or `to_addrs` is `None`, `send_message` fills those arguments with addresses extracted from the headers of `msg` as specified in **RFC 5322**: `from_addr` is set to the **Sender** field if it is present, and otherwise to the **From** field. **to_addrs** combines the values (if any) of the **To**, **Cc**, and **Bcc** fields from `msg`. If exactly one set of **Resent-** headers appear in the message, the regular headers are ignored and the **Resent-** headers are used instead. If the message contains more than one set of **Resent-** headers, a **ValueError** is raised, since there is no way to unambiguously detect the most recent set of **Resent-** headers.

`send_message` serializes `msg` using `BytesGenerator` with \r\n as the `linesep`, and calls `sendmail()` to transmit the resulting message. Regardless of the values of `from_addr` and `to_addrs`, `send_message` does not transmit any **Bcc** or **Resent-Bcc** headers that may appear in `msg`. If any of the addresses in `from_addr` and `to_addrs` contain non-ASCII characters and the server does not advertise SMTPUTF8 support, an **SMTPNotSupported** error is raised. Otherwise the `Message` is serialized with a clone of its `policy` with the `utf8` attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to `mail_options`.

*New in version 3.2.*
New in version 3.5: Support for internationalized addresses (SMTPUTF8).

SMTP.quit()

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.
SMTP Example

This example prompts the user for addresses needed in the message envelope (‘To’ and ‘From’ addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the RFC 822 headers. In particular, the ‘To’ and ‘From’ addresses must be included in the message headers explicitly.

```python
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
```
server.quit()

**Note:** In general, you will want to use the `email` package's features to construct an email message, which you can then send via `send_message()`; see `email: Examples`. 
smtpd — SMTP Server

Source code: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers.

See also: The aiosmtpd package is a recommended replacement for this module. It is based on asyncio and provides a more straightforward API. smtpd should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports RFC 5321, plus the RFC 1870 SIZE and RFC 6531 SMTPUTF8 extensions.
SMTPServer Objects

class smtplib.SMTPServer(localaddr, remoteaddr,
data_size_limit=33554432, map=None, enable_SMTPUTF8=False,
decode_data=False)

Create a new SMTPServer object, which binds to local address localaddr. It will treat remoteaddr as an upstream SMTP relayer. Both localaddr and remoteaddr should be a (host, port) tuple. The object inherits from asyncore.dispatcher, and so will insert itself into asyncore’s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the asyncore global socket map is used.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in RFC 6531) should be enabled. The default is False. When True, SMTPUTF8 is accepted as a parameter to the MAIL command and when present is passed to process_message() in the kwargs['mail_options'] list. decode_data and enable_SMTPUTF8 cannot be set to True at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When decode_data is False (the default), the server advertises the 8BITMIME extension (RFC 6152), accepts the BODY=8BITMIME parameter to
the MAIL command, and when present passes it to `process_message()` in the kwargs['mail_options'] list. `decode_data` and `enable_SMTPUTF8` cannot be set to True at the same time.

**process_message**(peer, mailfrom, rcpttos, data, **kwargs)

Raise a `NotImplementedError` exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as `remoteaddr` will be available as the `_remoteaddr` attribute. `peer` is the remote host's address, `mailfrom` is the envelope originator, `rcpttos` are the envelope recipients and `data` is a string containing the contents of the e-mail (which should be in RFC 5321 format).

If the `decode_data` constructor keyword is set to True, the `data` argument will be a unicode string. If it is set to False, it will be a bytes object.

`kwargs` is a dictionary containing additional information. It is empty if decode_data=True was given as an init argument, otherwise it contains the following keys:

- **mail_options**: a list of all received parameters to the MAIL command (the elements are uppercase strings; example: ['BODY=8BITMIME', 'SMTPUTF8']).
- **rcpt_options**: same as `mail_options` but for the RCPT command. Currently no RCPT TO options are supported, so for now this will always be an empty list.

Implementations of `process_message` should use the
**kwargs signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return None to request a normal 250 0k response; otherwise return the desired response string in RFC 5321 format.

channel_class

Override this in subclasses to use a custom SMTPChannel for managing SMTP clients.

New in version 3.4: The map constructor argument.

Changed in version 3.5: localaddr and remoteaddr may now contain IPv6 addresses.

New in version 3.5: The decode_data and enable_SMTPUTF8 constructor parameters, and the kwargs parameter to process_message() when decode_data is False.

Changed in version 3.6: decode_data is now False by default.
DebuggingServer Objects

class smtplib.**DebuggingServer**(*localaddr*, *remoteaddr*)

Create a new debugging server. Arguments are as per **SMTPServer**. Messages will be discarded, and printed on stdout.
PureProxy Objects

class smt pd. **PureProxy**(localadd r, remoteadd r)
Create a new pure proxy server. Arguments are as per **SMTPServer**. Everything will be relayed to remoteaddr. Note that running this has a good chance to make you into an open relay, so please be careful.
MailmanProxy Objects

```
class smtpd.MailmanProxy(localaddr, remoteaddr)
  Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.
```
SMTPChannel Objects

```python
class smtplib.SMTPChannel(server, conn, addr,
data_size_limit=33554432, map=None, enable_SMTPUTF8=False,
decode_data=False)
```

Create a new `SMTPChannel` object which manages the communication between the server and a single SMTP client.

*conn* and *addr* are as per the instance variables described below.

*data_size_limit* specifies the maximum number of bytes that will be accepted in a DATA command. A value of `None` or `0` means no limit.

*enable_SMTPUTF8* determines whether the SMTPUTF8 extension (as defined in [RFC 6531](https://tools.ietf.org/html/rfc6531)) should be enabled. The default is `False`. `decode_data` and *enable_SMTPUTF8* cannot be set to `True` at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

*decode_data* specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is `False`. `decode_data` and *enable_SMTPUTF8* cannot be set to `True` at the same time.

To use a custom SMTPChannel implementation you need to override the `SMTPServer.channel_class` of your `SMTPServer`.

*Changed in version 3.5:* The `decode_data` and *enable_SMTPUTF8*
parameters were added.

*Changed in version 3.6: decode_data is now False by default.*

The `SMTPChannel` has the following instance variables:

**smtp_server**
- Holds the `SMTPServer` that spawned this channel.

**conn**
- Holds the socket object connecting to the client.

**addr**
- Holds the address of the client, the second value returned by `socket.accept`

**received_lines**
- Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their `\r\n` line ending translated to `\n`.

**smtp_state**
- Holds the current state of the channel. This will be either COMMAND initially and then DATA after the client sends a “DATA” line.

**seen_greeting**
- Holds a string containing the greeting sent by the client in its “HELO”.

**mailfrom**
- Holds a string containing the address identified in the “MAIL FROM:” line from the client.
**rcpttos**
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

**received_data**
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating "\r\n\r\n".

**fqdn**
Holds the fully-qualified domain name of the server as returned by `socket.getfqdn()`.

**peer**
Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is `conn`.

The **SMTPChannel** operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base **SMTPChannel** class are methods for handling the following commands (and responding to them appropriately):

<table>
<thead>
<tr>
<th>Command</th>
<th>Action taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELO</td>
<td>Accepts the greeting from the client and stores it in <code>seen_greeting</code>. Sets server to base command mode.</td>
</tr>
<tr>
<td>EHLO</td>
<td>Accepts the greeting from the client and stores it in <code>seen_greeting</code>. Sets server to extended command mode.</td>
</tr>
<tr>
<td>NOOP</td>
<td>Takes no action.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>QUIT</td>
<td>Closes the connection cleanly.</td>
</tr>
<tr>
<td>MAIL</td>
<td>Accepts the “MAIL FROM:” syntax and stores the supplied address as <code>mailfrom</code>. In extended command mode, accepts the <strong>RFC 1870</strong> <code>SIZE</code> attribute and responds appropriately based on the value of <code>data_size_limit</code>.</td>
</tr>
<tr>
<td>RCPT</td>
<td>Accepts the “RCPT TO:” syntax and stores the supplied addresses in the <code>rcpttos</code> list.</td>
</tr>
<tr>
<td>RSET</td>
<td>Resets the <code>mailfrom</code>, <code>rcpttos</code>, and <code>received_data</code>, but not the greeting.</td>
</tr>
<tr>
<td>DATA</td>
<td>Sets the internal state to <code>DATA</code> and stores remaining lines from the client in <code>received_data</code> until the terminator <code>&quot;\r\n \r\n&quot;</code> is received.</td>
</tr>
<tr>
<td>HELP</td>
<td>Returns minimal information on command syntax</td>
</tr>
<tr>
<td>VRFY</td>
<td>Returns code 252 (the server doesn’t know if the address is valid)</td>
</tr>
<tr>
<td>EXPN</td>
<td>Reports that the command is not implemented.</td>
</tr>
</tbody>
</table>
The **telnetlib** module provides a **Telnet** class that implements the Telnet protocol. See [RFC 854](https://tools.ietf.org/html/rfc854) for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

```python
class telnetlib.Telnet(host=None, port=0[, timeout])
```

**Telnet** represents a connection to a Telnet server. The instance is initially not connected by default; the `open()` method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).
Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise `EOFError` when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

A `Telnet` object is a context manager and can be used in a `with` statement. When the `with` block ends, the `close()` method is called:

```python
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...```

*Changed in version 3.6:* Context manager support added

**See also:**

- **RFC 854 - Telnet Protocol Specification**
  - Definition of the Telnet protocol.
Telnet Objects

Telnet instances have the following methods:

Telnet. **read_until**(expected, timeout=None)

Read until a given byte string, `expected`, is encountered or until `timeout` seconds have passed.

When no match is found, return whatever is available instead, possibly empty bytes. Raise **EOFError** if the connection is closed and no cooked data is available.

Telnet. **read_all**()

Read all data until EOF as bytes; block until connection closed.

Telnet. **read_some**()

Read at least one byte of cooked data unless EOF is hit. Return `b''` if EOF is hit. Block if no data is immediately available.

Telnet. **read_very_eager**()

Read everything that can be without blocking in I/O (eager).

Raise **EOFError** if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet. **read_eager**()

Read readily available data.

Raise **EOFError** if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.
Telnet. **read_lazy()**
Process and return data already in the queues (lazy).

Raise **EOFError** if connection closed and no data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet. **read_very_lazy()**
Return any data available in the cooked queue (very lazy).

Raise **EOFError** if connection closed and no data available. Return `b''` if no cooked data available otherwise. This method never blocks.

Telnet. **read_sb_data()**
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a `SE` command. This method never blocks.

Telnet. **open(host, port=0[, timeout])**
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

Telnet. **msg(msg, *args)**
Print a debug message when the debug level is $> 0$. If extra arguments are present, they are substituted in the message using the standard string formatting operator.
Telnet.**set_debuglevel**(debuglevel)
Set the debug level. The higher the value of `debuglevel`, the more debug output you get (on `sys.stdout`).

Telnet.**close**()
Close the connection.

Telnet.**get_socket**()
Return the socket object used internally.

Telnet.**fileno**()
Return the file descriptor of the socket object used internally.

Telnet.**write**(buffer)
Write a byte string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise *OSError* if the connection is closed.

  *Changed in version 3.3:* This method used to raise *socket.error*, which is now an alias of *OSError*.

Telnet.**interact**()
Interaction function, emulates a very dumb Telnet client.

Telnet.**mt_interact**()
Multithreaded version of **interact**().

Telnet.**expect**(list, timeout=None)
Read until one from a list of a regular expressions matches.

  The first argument is a list of regular expressions, either compiled (**regex objects**) or uncompiled (byte strings). The optional second argument is a timeout, in seconds; the default is to block
indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the bytes read up till and including the match.

If end of file is found and no bytes were read, raise `EOFError`. Otherwise, when nothing matches, return `(-1, None, data)` where `data` is the bytes received so far (may be empty bytes if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

**Telnet**. `set_option_negotiation_callback(callback)`

Each time a telnet option is read on the input flow, this `callback` (if set) is called with the following parameters: `callback(telnet socket, command (DO/DONT/WILL/WONT), option)`. No other action is done afterwards by telnetlib.
Telnet Example

A simple example illustrating typical use:

```python
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")

if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```
This module provides immutable UUID objects (the `UUID` class) and the functions `uuid1()`, `uuid3()`, `uuid4()`, `uuid5()` for generating version 1, 3, 4, and 5 UUIDs as specified in RFC 4122.

If all you want is a unique ID, you should probably call `uuid1()` or `uuid4()`. Note that `uuid1()` may compromise privacy since it creates a UUID containing the computer’s network address. `uuid4()` creates a random UUID.

Depending on support from the underlying platform, `uuid1()` may or may not return a “safe” UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of `UUID` have an `is_safe` attribute which relays any information about the UUID’s safety, using this enumeration:

```
class uuid.SafeUUID
    New in version 3.7.

    safe
        The UUID was generated by the platform in a multiprocessing-safe way.

    unsafe
        The UUID was not generated in a multiprocessing-safe way.
```
unknown

The platform does not provide information on whether the UUID was generated safely or not.

class uuid.UUID(hex=None, bytes=None, bytes_le=None, fields=None, int=None, version=None, *, is_safe=SafeUUID.unknown)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the bytes argument, a string of 16 bytes in little-endian order as the bytes_le argument, a tuple of six integers (32-bit time_low, 16-bit time_mid, 16-bit time_hi_version, 8-bit clock_seq_hi_variant, 8-bit clock_seq_low, 48-bit node) as the fields argument, or a single 128-bit integer as the int argument. When a string of hex digits is given, curly braces, hyphens, and aURN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56'
b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x
UUID(int=0x1234567812345678123456781234567812345678)
```

Exactly one of hex, bytes, bytes_le, fields, or int must be given. The version argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given hex, bytes, bytes_le, fields, or int.

Comparison of UUID objects are made by way of comparing their UUID.int attributes. Comparison with a non-UUID object raises a TypeError.
str(uuid) returns a string in the form 12345678-1234-5678-1234-567812345678 where the 32 hexadecimal digits represent the UUID.

**UUID** instances have these read-only attributes:

**UUID.** **bytes**

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

**UUID.** **bytes_le**

The UUID as a 16-byte string (with `time_low`, `time_mid`, and `time_hi_version` in little-endian byte order).

**UUID.** **fields**

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time_low</code></td>
<td>the first 32 bits of the UUID</td>
</tr>
<tr>
<td><code>time_mid</code></td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td><code>time_hi_version</code></td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td><code>clock_seq_hi_variant</code></td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td><code>clock_seq_low</code></td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td><code>node</code></td>
<td>the last 48 bits of the UUID</td>
</tr>
<tr>
<td><code>time</code></td>
<td>the 60-bit timestamp</td>
</tr>
<tr>
<td><code>clock_seq</code></td>
<td>the 14-bit sequence number</td>
</tr>
</tbody>
</table>

**UUID.** **hex**

The UUID as a 32-character hexadecimal string.
The UUID as a 128-bit integer.

The UUID as a URN as specified in RFC 4122.

The UUID variant, which determines the internal layout of the UUID. This will be one of the constants RESERVED_NCS, RFC_4122, RESERVED_MICROSOFT, or RESERVED_FUTURE.

The UUID version number (1 through 5, meaningful only when the variant is RFC_4122).

An enumeration of SafeUUID which indicates whether the platform generated the UUID in a multiprocessing-safe way.

New in version 3.7.

The uuid module defines the following functions:

Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with the multicast bit (least significant bit of the first octet) set to 1 as recommended in RFC 4122. “Hardware address” means the MAC address of a network interface. On a machine with multiple network interfaces, universally administered MAC addresses (i.e. where the second least significant bit of the first octet is unset) will be preferred over
locally administered MAC addresses, but with no other ordering guarantees.

*Changed in version 3.7:* Universally administered MAC addresses are preferred over locally administered MAC addresses, since the former are guaranteed to be globally unique, while the latter are not.

**uuid.uuid1(node=None, clock_seq=None)**
Generate a UUID from a host ID, sequence number, and the current time. If `node` is not given, `getnode()` is used to obtain the hardware address. If `clock_seq` is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

**uuid.uuid3(namespace, name)**
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

**uuid.uuid4()**
Generate a random UUID.

**uuid.uuid5(namespace, name)**
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The `uuid` module defines the following namespace identifiers for use with `uuid3()` or `uuid5()`.

**uuid.NAMESPACE_DNS**
When this namespace is specified, the `name` string is a fully-qualified domain name.

**uuid.NAMESPACE_URL**
When this namespace is specified, the *name* string is a URL.

**uuid.** **NAMESPACE_OID**

When this namespace is specified, the *name* string is an ISO OID.

**uuid.** **NAMESPACE_X500**

When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The **uuid** module defines the following constants for the possible values of the variant attribute:

**uuid.** **RESERVED_NCS**

Reserved for NCS compatibility.

**uuid.** **RFC_4122**

Specifies the UUID layout given in **RFC 4122**.

**uuid.** **RESERVED_MICROSOFT**

Reserved for Microsoft compatibility.

**uuid.** **RESERVED_FUTURE**

Reserved for future definition.

---

**See also:**

**RFC 4122** - A Universally Unique IDentifier (UUID) URN Namespace

This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.
Example

Here are some examples of typical usage of the `uuid` module:

```python
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace and name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-e8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace and name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9ae199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\xb0\xb1\xc0\xc1'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
```
socketserver — A framework for network servers

Source code: Lib/socketserver.py

The socketserver module simplifies the task of writing network servers.

There are four basic concrete server classes:

```python
class socketserver.TCPServer(server_address, RequestHandlerClass, bind_and_activate=True)
```

This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If `bind_and_activate` is true, the constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the BaseServer base class.

```python
class socketserver.UDPServer(server_address, RequestHandlerClass, bind_and_activate=True)
```

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for TCPServer.

```python
class socketserver.UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)

class socketserver.UnixDatagramServer(server_address, RequestHandlerClass, bind_and_activate=True)
```

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they’re not available
on non-Unix platforms. The parameters are the same as for TCPServer.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn’t suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server’s address and the request handler class. It is recommended to use the server in a with statement. Then call the handle_request() or serve_forever() method of the server object to process one or many requests. Finally, call server_close() to close the socket (unless you used a with statement).

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute daemon_threads, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all threads created by ThreadingMixIn have exited.
Server classes have the same external methods and attributes, no matter what network protocol they use.
Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------------+
| TCPServer |------->| UnixStreamServer |
+------------------+
| UDPServer |------->| UnixDatagramServer |
```

Note that UnixDatagramServer derives from UDPServer, not from UnixStreamServer — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

```python
class socketserver.ForkingMixIn
class socketserver.ThreadingMixIn
```

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, ThreadingUDPServer is created as follows:

```python
class ThreadingUDPServer(ThreadingMixIn, UDPServer)
```
The mix-in class comes first, since it overrides a method defined in `UDPServer`. Setting the various attributes also changes the behavior of the underlying server mechanism.

`ForkingMixIn` and the Forking classes mentioned below are only available on POSIX platforms that support `fork()`.

socketserver.ForkingMixIn.server_close() waits until all child processes complete, except if socketserver.ForkingMixIn.block_on_close attribute is false.

socketserver.ThreadingMixIn.server_close() waits until all non-daemon threads complete, except if socketserver.ThreadingMixIn.block_on_close attribute is false. Use daemonic threads by setting ThreadingMixIn.daemon_threads to `True` to not wait until threads complete.

*Changed in version 3.7:* socketserver.ForkingMixIn.server_close() and socketserver.ThreadingMixIn.server_close() now waits until all child processes and non-daemonic threads complete. Add a new socketserver.ForkingMixIn.block_on_close class attribute to opt-in for the pre-3.7 behaviour.

```python
class socketserver.ForkingTCPServer
class socketserver.ForkingUDPServer
class socketserver.ThreadingTCPServer
class socketserver.ThreadingUDPServer
```

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from
**BaseRequestHandler** and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses **StreamRequestHandler** or **DatagramRequestHandler**.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service “deaf” while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class `handle()` method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use **selectors**.
to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See asyncore for another way to manage this.
Server Objects

class socketserver.BaseServer(server_address, RequestHandlerClass)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective server_address and RequestHandlerClass attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to selectors, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: get_request(), verify_request(), and process_request(). If the user-provided handle() method of the handler class raises an exception, the server's handle_error() method will be called. If no request is received within timeout seconds, handle_timeout() will be called and handle_request() will return.

serve_forever(poll_interval=0.5)

Handle requests until an explicit shutdown() request. Poll for shutdown every poll_interval seconds. Ignores the timeout attribute. It also calls service_actions(), which may be used by a subclass or mixin to provide actions specific to a
given service. For example, the `ForkingMixIn` class uses `service_actions()` to clean up zombie child processes.

*Changed in version 3.3:* Added `service_actions` call to the `serve_forever` method.

**service_actions()**  
This is called in the `serve_forever()` loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

*New in version 3.3.*

**shutdown()**  
Tell the `serve_forever()` loop to stop and wait until it does.

**server_close()**  
Clean up the server. May be overridden.

**address_family**  
The family of protocols to which the server’s socket belongs. Common examples are `socket.AF_INET` and `socket.AF_UNIX`.

**RequestHandlerClass**  
The user-provided request handler class; an instance of this class is created for each request.

**server_address**  
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For Internet protocols, this is a tuple containing a string giving the address,
and an integer port number: ('127.0.0.1', 80), for example.

**socket**
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

**allow_reuse_address**
Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

**request_queue_size**
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a “Connection denied” error. The default value is usually 5, but this can be overridden by subclasses.

**socket_type**
The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two common values.

**timeout**
Timeout duration, measured in seconds, or `None` if no timeout is desired. If `handle_request()` receives no incoming requests within the timeout period, the `handle_timeout()` method is called.

There are various server methods that can be overridden by
subclasses of base server classes like \texttt{TCPServer}; these methods aren't useful to external users of the server object.

\textbf{finish\_request}(\texttt{request, client\_address})

Actually processes the request by instantiating \texttt{RequestHandlerClass} and calling its \texttt{handle()} method.

\textbf{get\_request()}

Must accept a request from the socket, and return a 2-tuple containing the \textit{new} socket object to be used to communicate with the client, and the client’s address.

\textbf{handle\_error}(\texttt{request, client\_address})

This function is called if the \texttt{handle()} method of a \texttt{RequestHandlerClass} instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

\textit{Changed in version 3.6}: Now only called for exceptions derived from the \texttt{Exception} class.

\textbf{handle\_timeout()}

This function is called when the \texttt{timeout} attribute has been set to a value other than \texttt{None} and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

\textbf{process\_request}(\texttt{request, client\_address})

Calls \texttt{finish\_request()} to create an instance of the \texttt{RequestHandlerClass}. If desired, this function can create a
new process or thread to handle the request; the ForkingMixIn and ThreadingMixIn classes do this.

server_activate()
Called by the server’s constructor to activate the server. The default behavior for a TCP server just invokes listen() on the server’s socket. May be overridden.

server_bind()
Called by the server’s constructor to bind the socket to the desired address. May be overridden.

verify_request(request, client_address)
Must return a Boolean value; if the value is True, the request will be processed, and if it's False, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns True.

Changed in version 3.6: Support for the context manager protocol was added. Exiting the context manager is equivalent to calling server_close().
Request Handler Objects

```python
class socketserver.BaseRequestHandler
```

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new `handle()` method, and can override any of the other methods. A new instance of the subclass is created for each request.

```python
setup()
```

Called before the `handle()` method to perform any initialization actions required. The default implementation does nothing.

```python
handle()
```

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

```python
finish()
```

Called after the `handle()` method to perform any clean-up actions required. The default implementation does nothing. If
setup() raises an exception, this function will not be called.

class socketserver.StreamRequestHandler
class socketserver.DatagramRequestHandler

These BaseRequestHandler subclasses override the setup() and finish() methods, and provide self.rfile and self.wfile attributes. The self.rfile and self.wfile attributes can be read or written, respectively, to get the request data or return data to the client.

The rfile attributes of both classes support the io.BufferedIOBase readable interface, and DatagramRequestHandler.wfile supports the io.BufferedIOBase writable interface.

Changed in version 3.6: StreamRequestHandler.wfile also supports the io.BufferedIOBase writable interface.
Examples

socketserver.TCPServer Example

This is the server side:

```python
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.
    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication with
    the client.
    """
    def handle(self):
        # self.request is the TCP socket connected to
        self.data = self.request.recv(1024).strip()
        print("{}".format(self.client_address))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler):
        # Activate the server; this will keep running
        # interrupt the program with Ctrl-C
        server.serve_forever()
```
An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```python
class MyTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of
        self.data = self.rfile.readline().strip()
        print("{} wrote:").format(self.client_address)
        print(self.data)
        # Likewise, self.wfile is a file-like object
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```python
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))
```
# Receive data from the server and shut down
received = str(sock.recv(1024), "utf-8")

print("Sent:    ".format(data))
print("Received: ".format(received))

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent: hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent: python is nice
Received: PYTHON IS NICE
```

**socketserver.UDPServer Example**

This is the server side:

```python
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class
    """
```
The code consists of a pair of data and client socket, since there is no connection the client address must be given explicitly when sending data back via sendto().

```python
def handle(self):
    data = self.request[0].strip()
    socket = self.request[1]
    print("{} wrote:",format(self.client_address)
    print(data)
    socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler)
        server.serve_forever()
```

This is the client side:

```python
import socket
import sys
HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has
# Instead, data is directly sent to the recipient via
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent: {}".format(data))
print("Received: {}".format(received))
```
The output of the example should look exactly like for the TCP server
example.

Asynchronous Mixins
To build asynchronous handlers, use the ThreadingMixIn and
ForkingMixIn classes.
An example for the ThreadingMixIn class:
import socket
import threading
import socketserver
class ThreadedTCPRequestHandler(socketserver.BaseRequestHand
def handle(self):
data = str(self.request.recv(1024), 'ascii')
cur_thread = threading.current_thread()
response = bytes("{}: {}".format(cur_thread.
self.request.sendall(response)
class ThreadedTCPServer(socketserver.ThreadingMixIn,
pass
def client(ip, port, message):
with socket.socket(socket.AF_INET, socket.SOCK_STREAM
sock.connect((ip, port))
sock.sendall(bytes(message, 'ascii'))
response = str(sock.recv(1024), 'ascii')
print("Received: {}".format(response))
if __name__ == "__main__":
# Port 0 means to select an arbitrary unused port
HOST, PORT = "localhost", 0


server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)

with server:
    ip, port = server.server_address

    # Start a thread with the server -- that thread
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread
    server_thread.daemon = True
    server_thread.start()

    print("Server loop running in thread:", server_thread)

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The **ForkingMixIn** class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support `fork()`.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
http.server — HTTP servers

Source code: Lib/http/server.py

This module defines classes for implementing HTTP servers (Web servers).

**Warning:** http.server is not recommended for production. It only implements basic security checks.

One class, HTTPServer, is a socketserver.TCPServer subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```python
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler,
        server_address = ('', 8000)):
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

```python
class http.server.HTTPServer(server_address, RequestHandlerClass):
    This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

class http.server.ThreadingHTTPServer(server_address, RequestHandlerClass):
    This class is identical to HTTPServer but uses threads to handle requests by using the ThreadingMixIn. This is useful to handle
web browsers pre-opening sockets, on which HTTPServer would wait indefinitely.

*New in version 3.7.*

The HTTPServer and ThreadingHTTPServer must be given a RequestHandlerClass on instantiation, of which this module provides three different variants:

```python
class http.server.BaseHTTPRequestHandler(request, client_address, server)
```

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method SPAM, the do_SPAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler has the following instance variables:

**client_address**

Contains a tuple of the form `(host, port)` referring to the client’s address.

**server**

Contains the server instance.
close_connection

Boolean that should be set before handle_one_request() returns, indicating if another request may be expected, or if the connection should be shut down.

requestline

Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by handle_one_request(). If no valid request line was processed, it should be set to the empty string.

command

Contains the command (request type). For example, 'GET'.

path

Contains the request path.

request_version

Contains the version string from the request. For example, 'HTTP/1.0'.

headers

Holds an instance of the class specified by the MessageClass class variable. This instance parses and manages the headers in the HTTP request. The parse_headers() function from http.client is used to parse the headers and it requires that the HTTP request provide a valid RFC 2822 style header.

rfile

An io.BufferedIOBase input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

*Changed in version 3.6:* This is an `io.BufferedIOBase` stream.

`BaseHTTPRequestHandler` has the following attributes:

**server_version**
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

**sys_version**
Contains the Python system version, in a form usable by the `version_string` method and the `server_version` class variable. For example, 'Python/1.4'.

**error_message_format**
Specifies a format string that should be used by `send_error()` method for building an error response to the client. The string is filled by default with variables from `responses` based on the status code that passed to `send_error()`.

**error_content_type**
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is 'text/html'.

**protocol_version**
This specifies the HTTP protocol version used in responses. If set to 'HTTP/1.1', the server will permit HTTP persistent connections; however, your server must then include an accurate Content-Length header (using send_header()) in all of its responses to clients. For backwards compatibility, the setting defaults to 'HTTP/1.0'.

**MessageClass**

Specifies an email.message.Message-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to http.client.HTTPMessage.

**responses**

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, {code: (shortmessage, longmessage)}. The shortmessage is usually used as the message key in an error response, and longmessage as the explain key. It is used by send_response_only() and send_error() methods.

A BaseHTTPRequestHandler instance has the following methods:

**handle()**

Calls handle_one_request() once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate do_*() methods.

**handle_one_request()**

This method will parse and dispatch the request to the appropriate do_*() method. You should never need to
handle_expect_100()

When a HTTP/1.1 compliant server receives an Expect: 100-continue request header it responds back with a 100 Continue followed by 200 OK headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can choose to send 417 Expectation Failed as a response header and return False.

New in version 3.2.

send_error(code, message=None, explain=None)

Sends and logs a complete error reply to the client. The numeric code specifies the HTTP error code, with message as an optional, short, human readable description of the error. The explain argument can be used to provide more detailed information about the error; it will be formatted using the error_message_format attribute and emitted, after a complete set of headers, as the response body. The responses attribute holds the default values for message and explain that will be used if no value is provided; for unknown codes the default value for both is the string ???. The body will be empty if the method is HEAD or the response code is one of the following: 1xx, 204 No Content, 205 Reset Content, 304 Not Modified.

Changed in version 3.4: The error response includes a Content-Length header. Added the explain argument.

send_response(code, message=None)
Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by Server and Date headers. The values for these two headers are picked up from the version_string() and date_time_string() methods, respectively. If the server does not intend to send any other headers using the send_header() method, then send_response() should be followed by an end_headers() call.

*Changed in version 3.3:* Headers are stored to an internal buffer and end_headers() needs to be called explicitly.

**send_header**(keyword, value)

Adds the HTTP header to an internal buffer which will be written to the output stream when either end_headers() or flush_headers() is invoked. keyword should specify the header keyword, with value specifying its value. Note that, after the send_header calls are done, end_headers() MUST BE called in order to complete the operation.

*Changed in version 3.2:* Headers are stored in an internal buffer.

**send_response_only**(code, message=None)

Sends the response header only, used for the purposes when 100 Continue response is sent by the server to the client. The headers not buffered and sent directly the output stream. If the message is not specified, the HTTP message corresponding the response code is sent.

*New in version 3.2.*
**end_headers()**

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls `flush_headers()`.

*Changed in version 3.2:* The buffered headers are written to the output stream.

**flush_headers()**

Finally send the headers to the output stream and flush the internal headers buffer.

*New in version 3.3.*

**log_request**(code='-', size='-')

Logs an accepted (successful) request. `code` should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the `size` parameter.

**log_error**(...

Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (`format` and additional values).

**log_message**(format, ...)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The `format` argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

**version_string()**
Returns the server software’s version string. This is a combination of the `server_version` and `sys_version` attributes.

**date_time_string**(timestamp=None)

Returns the date and time given by `timestamp` (which must be `None` or in the format returned by `time.time()`), formatted for a message header. If `timestamp` is omitted, it uses the current date and time.

The result looks like `Sun, 06 Nov 1994 08:49:37 GMT`.

**log_date_time_string()**

Returns the current date and time, formatted for logging.

**address_string()**

Returns the client address.

*Changed in version 3.3:* Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.SimpleHTTPRequestHandler(request, client_address, server, directory=None)

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class `BaseHTTPRequestHandler`. This class implements the `do_GET()` and `do_HEAD()` functions.

The following are defined as class-level attributes of `SimpleHTTPRequestHandler`:
**server_version**

This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

**extensions_map**

A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

**directory**

If not specified, the directory to serve is the current working directory.

The SimpleHTTPRequestHandler class defines the following methods:

**do_HEAD()**

This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the do_GET() method for a more complete explanation of the possible headers.

**do_GET()**

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses
os.listdir() to scan the directory, and returns a 404 error response if the.listdir() fails.

If the request was mapped to a file, it is opened. Any OSError exception in opening the requested file is mapped to a 404, 'File not found' error. If there was a 'If-Modified-Since' header in the request, and the file was not modified after this time, a 304, 'Not Modified' response is sent. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the extensions_map variable, and the file contents are returned.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the test() function invocation in the http.server module.

*Changed in version 3.7:* Support of the 'If-Modified-Since' header.

The SimpleHTTPRequestHandler class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:
```python
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(('', PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

`http.server` can also be invoked directly using the `-m` switch of the interpreter with a `port` number argument. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

*New in version 3.4:* `--bind` argument was introduced.

By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

*New in version 3.7:* `--directory` specify alternate directory
```python
class http.server.CGIHTTPRequestHandler(request, client_address, server)

This class is used to serve either files or output of CGI scripts from
the current directory and below. Note that mapping HTTP
hierarchic structure to local directory structure is exactly as in
SimpleHTTPRequestHandler.

**Note:** CGI scripts run by the CGIHTTPRequestHandler class
cannot execute redirects (HTTP code 302), because code 200
(script output follows) is sent prior to execution of the CGI script.
This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a
file, if it guesses it to be a CGI script. Only directory-based CGI are
used — the other common server configuration is to treat special
extensions as denoting CGI scripts.

The do_GET() and do_HEAD() functions are modified to run CGI
scripts and serve the output, instead of serving files, if the request
leads to somewhere below the cgi_directories path.

The CGIHTTPRequestHandler defines the following data
member:

**cgi_directories**

This defaults to ['/cgi-bin', '/htbin'] and describes
directories to treat as containing CGI scripts.

The CGIHTTPRequestHandler defines the following method:

**do_POST()**

This method serves the 'POST' request type, only allowed for
CGI scripts. Error 501, “Can only POST to CGI scripts”, is
output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

**CGIHTTPRequestHandler** can be enabled in the command line by passing the **--cgi** option:

```
python -m http.server --cgi 8000
```
http.cookies — HTTP state management

Source code: Lib/http/cookies.py

The http.cookies module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the RFC 2109 and RFC 2068 specifications. It has since been discovered that MSIE 3.0x doesn’t follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, string.ascii_letters, string.digits and !#$%&'*+-.^_`|~: denote the set of valid characters allowed by this module in Cookie name (as key).

Changed in version 3.3: Allowed ‘:’ as a valid Cookie name character.

Note: On encountering an invalid cookie, CookieError is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch CookieError on parsing.

exception http.cookies.CookieError

Exception failing because of RFC 2109 invalidity: incorrect
attributes, incorrect Set-Cookie header, etc.

```python
class http.cookies.BaseCookie([input])
```
This class is a dictionary-like object whose keys are strings and whose values are `Morsel` instances. Note that upon setting a key to a value, the value is first converted to a `Morsel` containing the key and the value.

If `input` is given, it is passed to the `load()` method.

```python
class http.cookies.SimpleCookie([input])
```
This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()` to be the identity and `str()` respectively.

See also:

- **Module `http.cookiejar`**
  HTTP cookie handling for web `clients`. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

- **RFC 2109 - HTTP State Management Mechanism**
  This is the state management specification implemented by this module.
Cookie Objects

BaseCookie.value_decode(val)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in BaseCookie — it exists so it can be overridden.

BaseCookie.value_encode(val)
Return an encoded value. val can be any type, but return value must be a string. This method does nothing in BaseCookie — it exists so it can be overridden.

In general, it should be the case that value_encode() and value_decode() are inverses on the range of value_decode.

BaseCookie.output(attrs=None, header='Set-Cookie:', sep='
')
Return a string representation suitable to be sent as HTTP headers. attrs and header are sent to each Morsel's output() method. sep is used to join the headers together, and is by default the combination '\r\n' (CRLF).

BaseCookie.js_output(attrs=None)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for attrs is the same as in output().

BaseCookie.load(rawdata)
If rawdata is a string, parse it as an HTTP_COOKIE and add the values found there as Morsels. If it is a dictionary, it is equivalent
for k, v in rawdata.items():
    cookie[k] = v
Morsel Objects

```python
class http.cookies.Morsel
```

Abstract a key/value pair, which has some RFC 2109 attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid RFC 2109 attributes, which are

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`

The attribute `httponly` specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive and their default value is `''`.

*Changed in version 3.5:* `__eq__()` now takes `key` and `value` into account.

*Changed in version 3.7:* Attributes `key`, `value` and `coded_value` are read-only. Use `set()` for setting them.

**Morsel.value**

The value of the cookie.
Morsel.coded_value
   The encoded value of the cookie — this is what should be sent.

Morsel.key
   The name of the cookie.

Morsel.set(key, value, coded_value)
   Set the key, value and coded_value attributes.

Morsel.isReservedKey(K)
   Whether K is a member of the set of keys of a Morsel.

Morsel.output(attrs=None, header='Set-Cookie:'
   Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default "Set-Cookie:".

Morsel.js_output(attrs=None)
   Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

   The meaning for attrs is the same as in output().

Morsel.OutputString(attrs=None)
   Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

   The meaning for attrs is the same as in output().

Morsel.update(values)
   Update the values in the Morsel dictionary with the values in the
dictionary values. Raise an error if any of the keys in the values dict is not a valid RFC 2109 attribute.

*Changed in version 3.5:* an error is raised for invalid keys.

Morsel. `copy(value)`
Return a shallow copy of the Morsel object.

*Changed in version 3.5:* return a Morsel object instead of a dict.

Morsel. `setdefault(key, value=None)`
Raise an error if key is not a valid RFC 2109 attribute, otherwise behave the same as `dict.setdefault()`.
Example

The following example demonstrates how to use the `http.cookies` module.

```python
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C)  # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output())  # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger")  # load from
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L="Loves"; fudge")
>>> print(C)
Set-Cookie: keebler="E=everybody; L="Loves"; fudge
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
```
Set-Cookie: oreo=doublestuff; Path=/

```python
>>> C = cookies.SimpleCookie()

>>> C["twix"] = "none for you"

>>> C["twix"]').value

'none for you'

>>> C = cookies.SimpleCookie()

>>> C["number"] = 7  # equivalent to C["number"] = str(7)

>>> C["string"] = "seven"

>>> C["number"]').value

'7'

>>> C["string"]').value

'seven'

>>> print(C)

Set-Cookie: number=7
Set-Cookie: string=seven
```
http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The http.cookiejar module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – cookies – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by RFC 2965 are handled. RFC 2965 handling is switched off by default. RFC 2109 cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. http.cookiejar attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the max-age and port cookie-attributes introduced with RFC 2965.

Note: The various named parameters found in Set-Cookie and Set-Cookie2 headers (eg. domain and expires) are conventionally referred to as attributes. To distinguish them from Python attributes, the documentation for this module uses the term cookie-attribute instead.

The module defines the following exception:
exception http.cookiejar.LoadError

Instances of FileCookieJar raise this exception on failure to load cookies from a file. LoadError is a subclass of OSError.

Changed in version 3.3: LoadError was made a subclass of OSError instead of IOError.

The following classes are provided:

class http.cookiejar.CookieJar(policy=None)

policy is an object implementing the CookiePolicy interface.

The CookieJar class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. CookieJar instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.FileCookieJar(filename, delayload=None, policy=None)

policy is an object implementing the CookiePolicy interface. For the other arguments, see the documentation for the corresponding attributes.

A CookieJar which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are NOT loaded from the named file until either the load() or revert() method is called. Subclasses of this class are documented in section FileCookieJar subclasses and co-operation with web browsers.

class http.cookiejar.CookiePolicy

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.
class http.cookiejar.DefaultCookiePolicy(blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=http.cookiejar.DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False)

Constructor arguments should be passed as keyword arguments only. **blocked_domains** is a sequence of domain names that we never accept cookies from, nor return cookies to. **allowed_domains** if not *None*, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for CookiePolicy and DefaultCookiePolicy objects.

**DefaultCookiePolicy** implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (i.e. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or **rfc2109_as_netscape** is *True*, RFC 2109 cookies are ‘downgraded’ by the CookieJar instance to Netscape cookies, by setting the version attribute of the Cookie instance to 0. **DefaultCookiePolicy** also provides some parameters to allow some fine-tuning of policy.

class http.cookiejar.Cookie

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of http.cookiejar construct their own Cookie instances. Instead, if necessary, call make_cookies() on a CookieJar instance.
See also:

Module `urllib.request`
URL opening with automatic cookie handling.

Module `http.cookies`
HTTP cookie classes, principally useful for server-side code. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html
The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the ‘Netscape cookie protocol’ implemented by all the major browsers (and `http.cookiejar`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism
Obsoleted by RFC 2965. Uses `Set-Cookie` with version=1.

RFC 2965 - HTTP State Management Mechanism
The Netscape protocol with the bugs fixed. Uses `Set-Cookie2` in place of `Set-Cookie`. Not widely used.

http://kristol.org/cookie/errata.html
Unfinished errata to RFC 2965.

RFC 2964 - Use of HTTP State Management
CookieJar and FileCookieJar Objects

CookieJar objects support the iterator protocol for iterating over contained Cookie objects.

CookieJar has the following methods:

CookieJar.add_cookie_header(request)
Add correct Cookie header to request.

If policy allows (ie. the rfc2965 and hide_cookie2 attributes of the CookieJar's CookiePolicy instance are true and false respectively), the Cookie2 header is also added when appropriate.

The request object (usually a urllib.request.Request instance) must support the methods get_full_url(), get_host(), get_type(), unverifiable(), has_header(), get_header(), header_items(), add_unredirected_header() and origin_req_host attribute as documented by urllib.request.

Changed in version 3.3: request object needs origin_req_host attribute. Dependency on a deprecated method get_origin_req_host() has been removed.

CookieJar.extract_cookies(response, request)
Extract cookies from HTTP response and store them in the CookieJar, where allowed by policy.

The CookieJar will look for allowable Set-Cookie and Set-Cookie2 headers in the response argument, and store cookies as
appropriate (subject to the `CookiePolicy.set_ok()` method's approval).

The `response` object (usually the result of a call to `urllib.request.urlopen()`, or similar) should support an `info()` method, which returns an `email.message.Message` instance.

The `request` object (usually a `urllib.request.Request` instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by `urllib.request`. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

*Changed in version 3.3: `request` object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.*

`CookieJar.set_policy(policy)`

Set the `CookiePolicy` instance to be used.

`CookieJar.make_cookies(response, request)`

Return sequence of `Cookie` objects extracted from `response` object.

See the documentation for `extract_cookies()` for the interfaces required of the `response` and `request` arguments.

`CookieJar.set_cookie_if_ok(cookie, request)`

Set a `Cookie` if policy says it’s OK to do so.

`CookieJar.set_cookie(cookie)`
Set a **Cookie**, without checking with policy to see whether or not it should be set.

**CookieJar**. `clear([domain[, path[, name]]])`

Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that `domain` will be removed. If given two arguments, cookies belonging to the specified `domain` and URL `path` are removed. If given three arguments, then the cookie with the specified `domain`, `path` and `name` is removed.

Raises **KeyError** if no matching cookie exists.

**CookieJar**. `clear_session_cookies()`

Discard all session cookies.

Discards all contained cookies that have a true discard attribute (usually because they had either no max-age or expires cookie-attribute, or an explicit discard cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true `ignore_discard` argument.

**FileCookieJar** implements the following additional methods:

**FileCookieJar**. `save(filename=None, ignore_discard=False, ignore.expires=False)`

Save cookies to a file.
This base class raises `NotImplementedError`. Subclasses may leave this method unimplemented.

`filename` is the name of file in which to save cookies. If `filename` is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is `None`, `ValueError` is raised.

`ignore_discard`: save even cookies set to be discarded.
`ignore_expires`: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the `load()` or `revert()` methods.

FileCookieJar. `load(filename=None, ignore_discard=False, ignore_expires=False)`
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `OSError` may be raised, for example if the file does not exist.

*Changed in version 3.3*: `IOError` used to be raised, it is now an alias of `OSError`.

FileCookieJar. `revert(filename=None, ignore_discard=False, ignore_expires=False)`
Clear all cookies and reload cookies from a saved file.
revert() can raise the same exceptions as load(). If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar. **filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar. **delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A CookieJar object may ignore it. None of the FileCookieJar classes included in the standard library lazily loads cookies.
FileCookieJar subclasses and cooperation with web browsers

The following CookieJar subclasses are provided for reading and writing.

class http.cookiejar.MozillaCookieJar(filename, delayload=None, policy=None)

A FileCookieJar that can load from and save cookies to disk in the Mozilla cookies.txt file format (which is also used by the Lynx and Netscape browsers).

Note: This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-attributes such as port.

Warning: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.LWPCookieJar(filename, delayload=None, policy=None)

A FileCookieJar that can load from and save cookies to disk in format compatible with the libwww-perl library's Set-Cookie3 file format. This is convenient if you want to store cookies in a human-readable file.
CookiePolicy Objects

Objects implementing the CookiePolicy interface have the following methods:

CookiePolicy.set_ok(cookie, request)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.extract_cookies().

CookiePolicy.return_ok(cookie, request)
Return boolean value indicating whether cookie should be returned to server.

cookie is a Cookie instance. request is an object implementing the interface defined by the documentation for CookieJar.add_cookie_header().

CookiePolicy.domain_return_ok(domain, request)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from domain_return_ok() and path_return_ok() leaves all the work to return_ok().

If domain_return_ok() returns true for the cookie domain, path_return_ok() is called for the cookie path. Otherwise,
`path_return_ok()` and `return_ok()` are never called for that cookie domain. If `path_return_ok()` returns true, `return_ok()` is called with the `Cookie` object itself for a full check. Otherwise, `return_ok()` is never called for that cookie path.

Note that `domain_return_ok()` is called for every cookie domain, not just for the request domain. For example, the function might be called with both `.example.com` and `www.example.com` if the request domain is `www.example.com`. The same goes for `path_return_ok()`.

The `request` argument is as documented for `return_ok()`.

CookiePolicy.\texttt{path\_return\_ok}(\texttt{path, request})

Return false if cookies should not be returned, given cookie path.

See the documentation for `domain_return_ok()`.

In addition to implementing the methods above, implementations of the `CookiePolicy` interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.\texttt{netscape}

Implement Netscape protocol.

CookiePolicy.\texttt{rfc2965}

Implement RFC 2965 protocol.

CookiePolicy.\texttt{hide\_cookie2}

Don’t add Cookie2 header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).
The most useful way to define a `CookiePolicy` class is by subclassing from `DefaultCookiePolicy` and overriding some or all of the methods above. `CookiePolicy` itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).
DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```python
import http.cookiejar

class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the CookiePolicy interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the blocked_domains constructor argument, and blocked_domains() and set_blocked_domains() methods (and the corresponding argument and methods for allowed_domains). If you set a whitelist,
you can turn it off again by setting it to None.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains "192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.blocked_domains()
Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.set_blocked_domains(blocked_domains)
Set the sequence of blocked domains.

DefaultCookiePolicy.is_blocked(domain)
Return whether domain is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.allowed_domains()
Return None, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.set_allowed_domains(allowed_domains)
Set the sequence of allowed domains, or None.

DefaultCookiePolicy.is_not_allowed(domain)
Return whether *domain* is not on the whitelist for setting or receiving cookies.

**DefaultCookiePolicy** instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

**DefaultCookiePolicy.**`rfc2109_as_netscape` If true, request that the CookieJar instance downgrade RFC 2109 cookies (i.e. cookies received in a Set-Cookie header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the Cookie instance to 0. The default value is `None`, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

**DefaultCookiePolicy.**`strict_domain` Don’t allow sites to set two-component domains with country-code top-level domains like .co.uk, .gov.uk, .co.nz etc. This is far from perfect and isn’t guaranteed to work!

**RFC 2965** protocol strictness switches:

**DefaultCookiePolicy.**`strict_rfc2965_unverifiable` Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are never blocked on the basis of verifiability

Netscape protocol strictness switches:
DefaultCookiePolicy. **strict_ns_unverifiable**

Apply RFC 2965 rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy. **strict_ns_domain**

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy. **strict_ns_set_initial_dollar**

Ignore cookies in Set-Cookie: headers that have names starting with '$'.

DefaultCookiePolicy. **strict_ns_set_path**

Don’t allow setting cookies whose path doesn’t path-match request URI.

strict_ns_domain is a collection of flags. Its value is constructed by or-ing together (for example, DomainStrictNoDots|DomainStrictNonDomain means both flags are set).

DefaultCookiePolicy. **DomainStrictNoDots**

When setting cookies, the ‘host prefix’ must not contain a dot (eg. www.foo.bar.com can’t set a cookie for .bar.com, because www.foo contains a dot).

DefaultCookiePolicy. **DomainStrictNonDomain**

Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. spam.example.com won’t be returned cookies from example.com that had no domain cookie-attribute).

DefaultCookiePolicy. **DomainRFC2965Match**
When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

**DefaultCookiePolicy. DomainLiberal**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

**DefaultCookiePolicy. DomainStrict**
Equivalent to DomainStrictNoDots|DomainStrictNonDomain.
Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the max-age and expires cookie-attributes contain equivalent information, and because RFC 2109 cookies may be ‘downgraded’ by http.cookiejar from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a CookiePolicy method. The class does not enforce internal consistency, so you should know what you’re doing if you do that.

**Cookie.version**

Integer or None. Netscape cookies have version 0. RFC 2965 and RFC 2109 cookies have a version cookie-attribute of 1. However, note that http.cookiejar may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case version is 0.

**Cookie.name**

Cookie name (a string).

**Cookie.value**

Cookie value (a string), or None.

**Cookie.port**

String representing a port or a set of ports (eg. ‘80’, or ‘80,8080’), or None.

**Cookie.path**
Cookie path (a string, eg. '/acme/rocket_launchers').

Cookie. **secure**
   True if cookie should only be returned over a secure connection.

Cookie. **expires**
   Integer expiry date in seconds since epoch, or **None**. See also the `is_expired()` method.

Cookie. **discard**
   True if this is a session cookie.

Cookie. **comment**
   String comment from the server explaining the function of this cookie, or **None**.

Cookie. **comment_url**
   URL linking to a comment from the server explaining the function of this cookie, or **None**.

Cookie. **rfc2109**
   True if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a `Set-Cookie` header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because `http.cookiejar` may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

Cookie. **port_specified**
   True if a port or set of ports was explicitly specified by the server (in the `Set-Cookie / Set-Cookie2` header).

Cookie. **domain_specified**
True if a domain was explicitly specified by the server.

Cookie. **domain_initial_dot**
True if the domain explicitly specified by the server began with a dot (').

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie. **has_nonstandard_attr**(*name*)
Return true if cookie has the named cookie-attribute.

Cookie. **get_nonstandard_attr**(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie. **set_nonstandard_attr**(*name*, *value*)
Set the value of the named cookie-attribute.

The **Cookie** class also defines the following method:

Cookie. **is_expired**(*now=None*)
True if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.
Examples

The first example shows the most common usage of http.cookiejar:

```python
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```python
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".nets	opener = urllib.request.build_opener(urllib.request.
r = opener.open("http://example.com/")
```

The next example illustrates the use of DefaultCookiePolicy. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```python
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.
```python
r = opener.open("http://example.com/")
```
xmlrpc — XMLRPC server and client modules

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data.

xmlrpc is a package that collects server and client modules implementing XML-RPC. The modules are:

- `xmlrpc.client`
- `xmlrpc.server`
xmlrpc.client — XML-RPC client access

Source code: Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning: The xmlrpc.client module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Changed in version 3.5: For HTTPS URIs, xmlrpc.client now performs all the necessary certificate and hostname checks by default.

class xmlrpc.client.ServerProxy(uri, transport=None, encoding=None, verbose=False, allow_none=False, use_datetime=False, use_builtin_types=False, *, context=None)

    Changed in version 3.3: The use_builtin_types flag was added.

A ServerProxy instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal
SafeTransport instance for https: URLs and an internal HTTP Transport instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If allow_none is true, the Python constant None will be translated into XML; the default behaviour is for None to raise a TypeError. This is a commonly-used extension to the XML-RPC specification, but isn’t supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The use_builtin_types flag can be used to cause date/time values to be presented as datetime.datetime objects and binary data to be presented as bytes objects; this flag is false by default. datetime.datetime, bytes and bytearray objects may be passed to calls. The obsolete use_datetime flag is similar to use_builtin_types but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: http://user:pass@host:port/path. The user:pass portion will be base64-encoded as an HTTP ‘Authorization’ header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, context may be ssl.SSLContext and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can
also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

<table>
<thead>
<tr>
<th>XML-RPC type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>int, i1, i2, i4, i8 or biginteger</td>
<td>int in range from -2147483648 to 2147483647. Values get the &lt;int&gt; tag.</td>
</tr>
<tr>
<td>double or float</td>
<td>float. Values get the &lt;double&gt; tag.</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>array</td>
<td>list or tuple containing conformable elements. Arrays are returned as lists.</td>
</tr>
<tr>
<td>struct</td>
<td>dict. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <strong>dict</strong> attribute is transmitted.</td>
</tr>
<tr>
<td>dateTime.iso8601</td>
<td>DateTime or datetime.datetime. Returned type depends on values of use_builtin_types and use_datetime flags.</td>
</tr>
<tr>
<td>base64</td>
<td>Binary, bytes or bytearray. Returned type depends on the value of the use_builtin_types flag.</td>
</tr>
<tr>
<td>nil</td>
<td>The None constant. Passing is allowed only if allow_none is true.</td>
</tr>
</tbody>
</table>
bigdecimal | decimal.Decimal. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special Fault instance, used to signal XML-RPC server errors, or ProtocolError used to signal an error in the HTTP/HTTPS transport layer. Both Fault and ProtocolError derive from a base class called Error. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as <, >, and & will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use bytes or bytearray classes or the Binary wrapper class described below.

Server is retained as an alias for ServerProxy for backwards compatibility. New code should use ServerProxy.

Changed in version 3.5: Added the context argument.

Changed in version 3.6: Added support of type tags with prefixes (e.g. ex:nil). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: i1, i2, i8, biginteger, float and bigdecimal. See http://ws.apache.org/xmlrpc/types.html for a description.

See also:
**XML-RPC HOWTO**

A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

**XML-RPC Introspection**

Describes the XML-RPC protocol extension for introspection.

**XML-RPC Specification**

The official specification.

**Unofficial XML-RPC Errata**

Fredrik Lundh’s “unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at ‘best practices’ to use when designing your own XML-RPC implementations.”
ServerProxy Objects

A ServerProxy instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a Fault or ProtocolError object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved system attribute:

ServerProxy.system.listMethods()
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

ServerProxy.system.methodSignature(name)
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.
If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(name)

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Changed in version 3.5: Instances of ServerProxy support the context manager protocol for closing the underlying transport.

A working example follows. The server code:

```python
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```
DateTime Objects

class xmlrpc.client.DateTime
   This class may be initialized with seconds since the epoch, a time
tuple, an ISO 8601 time/date string, or a datetime.datetime
instance. It has the following methods, supported mainly for
internal use by the marshalling/unmarshalling code:

   decode(string)
      Accept a string as the instance’s new time value.

   encode(out)
      Write the XML-RPC encoding of this DateTime item to the out
stream object.

   It also supports certain of Python’s built-in operators through rich
comparison and __repr__() methods.

A working example follows. The server code:

```python
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(('localhost', 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```
The client code for the preceding server:

```python
import xmlrpc.client
import datetime


today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y-%m-%d %H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```
Binary Objects

class xmlrpc.client.Binary

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

**data**

The binary data encapsulated by the Binary instance. The data is provided as a bytes object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

**decode**(bytes)

Accept a base64 bytes object and decode it as the instance’s new data.

**encode**(out)

Write the XML-RPC base 64 encoding of this binary item to the out stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python’s built-in operators through __eq__() and __ne__() methods.

Example usage of the binary objects. We’re going to transfer an image over XMLRPC:
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()

The client gets the image and saves it to a file:

import xmlrpc.client

with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
Fault Objects

class xmlrpc.client.Fault

A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a Fault by returning a complex type object. The server code:

```python
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(('localhost', 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpc.client
```
proxy = xmlrpc.client.ServerProxy("http://localhost:

try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
ProtocolError Objects

class xmlrpc.client.ProtocolError
A ProtocolError object describes a protocol error in the underlying transport layer (such as a 404 ‘not found’ error if the server named by the URI does not exist). It has the following attributes:

url
The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we’re going to intentionally cause a ProtocolError by providing an invalid URI:

```python
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
```
print("URL: %s" % err.url)
print("HTTP/HTTPS headers: %s" % err.headers)
print("Error code: %d" % err.errcode)
print("Error message: %s" % err.errmsg)
MultiCall Objects

The MultiCall object provides a way to encapsulate multiple calls to a remote server into a single request [1].

```python
class xmlrpc.client.MultiCall(server):
    Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.
```

A usage example of this class follows. The server code:

```python
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
```
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()

The client code for the preceding server:

```python
import xmlrpc.client

multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```
Convenience Functions

```python
xmlrpc.client.dumps(params, methodname=None, methodresponse=None, encoding=None, allow_none=False)
```

Convert `params` into an XML-RPC request. or into a response if `methodresponse` is true. `params` can be either a tuple of arguments or an instance of the `Fault` exception class. If `methodresponse` is true, only a single value can be returned, meaning that `params` must be of length 1. `encoding`, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python’s `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for `allow_none`.

```python
xmlrpc.client.loads(data, use_datetime=False, use_builtin_types=False)
```

Convert an XML-RPC request or response into Python objects, a `params, methodname`). `params` is a tuple of argument; `methodname` is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The `use_builtin_types` flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete `use_datetime` flag is similar to `use_builtin_types` but it applies only to date/time values.

*Changed in version 3.3:* The `use_builtin_types` flag was added.
Example of Client Usage

```python
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000")  # local server
with ServerProxy("http://betty.userland.com") as proxy:
    print(proxy)
    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through an HTTP proxy, you need to define a custom transport. The following example shows how:

```python
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):
    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
```
transport.set_proxy('proxy-server', 8080)

server = xmlrpc.client.ServerProxy('http://betty.userland.com')

print(server.examples.getStateName(41))
Example of Client and Server Usage

See SimpleXMLRPCServer Example.

Footnotes

[1] This approach has been first presented in a discussion on xmlrpc.com.
The `xmlrpc.server` module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using `SimpleXMLRPCServer`, or embedded in a CGI environment, using `CGIXMLRPCRequestHandler`.

**Warning:** The `xmlrpc.server` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

class `xmlrpc.server.SimpleXMLRPCServer`(
`addr`,
requestHandler=`SimpleXMLRPCRequestHandler`, logRequests=True,
allow_none=False, encoding=None, bind_and_activate=True,
use_builtin_types=False)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The `requestHandler` parameter should be a factory for request handler instances; it defaults to `SimpleXMLRPCRequestHandler`. The `addr` and `requestHandler` parameters are passed to the `socketserver.TCPServer` constructor. If `logRequests` is true (the default), requests will be logged; setting this parameter to false will turn off logging. The `allow_none` and `encoding` parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The `bind_and_activate` parameter
controls whether server_bind() and server_activate() are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the allow_reuse_address class variable before the address is bound. The use_builtin_types parameter is passed to the loads() function and controls which types are processed when date/times values or binary data are received; it defaults to false.

*Changed in version 3.3:* The use_builtin_types flag was added.

```python
class xmlrpc.server.CGIXMLRPCRequestHandler(allow_none=False, encoding=None, use_builtin_types=False)
```

Create a new instance to handle XML-RPC requests in a CGI environment. The allow_none and encoding parameters are passed on to xmlrpc.client and control the XML-RPC responses that will be returned from the server. The use_builtin_types parameter is passed to the loads() function and controls which types are processed when date/times values or binary data are received; it defaults to false.

*Changed in version 3.3:* The use_builtin_types flag was added.

```python
class xmlrpc.server.SimpleXMLRPCRequestHandler
```

Create a new request handler instance. This request handler supports POST requests and modifies logging so that the logRequests parameter to the SimpleXMLRPCServer constructor parameter is honored.
SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

`SimpleXMLRPCServer.register_function(function=None, name=None)`

Register a function that can respond to XML-RPC requests. If `name` is given, it will be the method name associated with `function`, otherwise `function.__name__` will be used. `name` is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, `name` can only be given as a keyword argument to register `function` under `name`. If no `name` is given, `function.__name__` will be used.

*Changed in version 3.7: `register_function()` can be used as a decorator.*

`SimpleXMLRPCServer.register_instance(instance, allow_dotted_names=False)`

Register an object which is used to expose method names which have not been registered using `register_function()`. If `instance` contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that `params` does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as
func(*params), expanding the parameter list. The return value from _dispatch() is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method.

If the optional allow_dotted_names argument is true and the instance does not have a _dispatch() method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**Warning:** Enabling the allow_dotted_names option allows intruders to access your module’s global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

SimpleXMLRPCServer.register_introspection_functions

Registers the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

SimpleXMLRPCServer.register_multicall_functions()

Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.rpc_paths

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 “no such page” HTTP error. If this tuple is empty, all paths will be considered valid. The default value is (('/', '/RPC2')).
SimpleXMLRPCServer Example

Server code:

```python
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
    requestHandler=RequestHandler,
) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y
    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```
The following client code will call the methods made available by the preceding server:

```python
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

register_function() can also be used as a decorator. The previous server example can register functions in a decorator way:

```python
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                         requestHandler=RequestHandler,
                         server.register_introspection_functions()  

    # Register pow() function; this will use the value
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
```
return x + y

# Register a function under function.__name__.
@server.register_function
def mul(x, y):
    return x * y

server.serve_forever()

The following example included in the Lib/xmlrpc/server.py module shows a server allowing dotted names and registering a multicall function.

**Warning:** Enabling the *allow_dotted_names* option allows intruders to access your module’s global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```python
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
```
print('Serving XML-RPC on localhost port 8000')
try:
    server.serve_forever()
except KeyboardInterrupt:
    print('
Keyboard interrupt received, exiting.
')
sys.exit(0)

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py`:

```
server = ServerProxy("http://localhost:8000")
try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2, 9)
multi.add(1, 2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```
CGIXMLRPCRequestHandler

The `CGIXMLRPCRequestHandler` class can be used to handle XML-RPC requests sent to Python CGI scripts.

`CGIXMLRPCRequestHandler.register_function(function=None, name=None)`

Register a function that can respond to XML-RPC requests. If `name` is given, it will be the method name associated with `function`, otherwise `function.__name__` will be used. `name` is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, `name` can only be given as a keyword argument to register `function` under `name`. If no `name` is given, `function.__name__` will be used.

*Changed in version 3.7:* `register_function()` can be used as a decorator.

`CGIXMLRPCRequestHandler.register_instance(instance)`

Register an object which is used to expose method names which have not been registered using `register_function()`. If `instance` contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If `instance` does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple
hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**CGIXMLRPCRequestHandler.register_introspection_functions**

Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

**CGIXMLRPCRequestHandler.register_multicall_functions**

Register the XML-RPC multicall function `system.multicall`.

**CGIXMLRPCRequestHandler.handle_request**

Handle an XML-RPC request. If `request_text` is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```python
class MyFuncs:
    def mul(self, x, y):
        return x * y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x, y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```
Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using `DocXMLRPCServer`, or embedded in a CGI environment, using `DocCGIXMLRPCRequestHandler`.

class xmlrpc.server.DocXMLRPCServer(addr, requestHandler=DocXMLRPCRequestHandler, logRequests=True, allow_none=False, encoding=None, bind_and_activate=True, use_builtin_types=True)

Create a new server instance. All parameters have the same meaning as for `SimpleXMLRPCServer`; `requestHandler` defaults to `DocXMLRPCRequestHandler`.

*Changed in version 3.3:* The `use_builtin_types` flag was added.

class xmlrpc.server.DocCGIXMLRPCRequestHandler

Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.DocXMLRPCRequestHandler

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the `logRequests` parameter to the `DocXMLRPCServer` constructor parameter is honored.
DocXMLRPCServer Objects

The DocXMLRPCServer class is derived from SimpleXMLRPCServer and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.set_server_title(server_title)
Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocXMLRPCServer.set_server_name(server_name)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocXMLRPCServer.set_server_documentation(server_doc)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
DocCGIXMLRPCRequestHandler

The DocCGIXMLRPCRequestHandler class is derived from CGIXMLRPCRequestHandler and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.set_server_title(server_title)
Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocCGIXMLRPCRequestHandler.set_server_name(server_name)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocCGIXMLRPCRequestHandler.set_server_documentation
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.
ipaddress — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see An introduction to the ipaddress module.

New in version 3.3.
Convenience factory functions

The `ipaddress` module provides factory functions to conveniently create IP addresses, networks and interfaces:

```python
ipaddress.ip_address(address)
```
Return an `IPv4Address` or `IPv6Address` object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A `ValueError` is raised if `address` does not represent a valid IPv4 or IPv6 address.

```python
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

```python
ipaddress.ip_network(address, strict=True)
```
Return an `IPv4Network` or `IPv6Network` object depending on the IP address passed as argument. `address` is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than 2**32 will be considered to be IPv4 by default. `strict` is passed to `IPv4Network` or `IPv6Network` constructor. A `ValueError` is raised if `address` does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```python
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

```python
ipaddress.ip_interface(address)
```
Return an `IPv4Interface` or `IPv6Interface` object depending
on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than $2^{32}$ will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.
IP Addresses

Address objects

The IPv4Address and IPv6Address objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by IPv4Address objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are hashable, so they can be used as keys in dictionaries.

```python
class ipaddress.IPv4Address(address)
    Construct an IPv4 address. An AddressValueError is raised if address is not a valid IPv4 address.
```

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. 192.168.0.1). Each integer represents an octet (byte) in the address. Leading zeroes are tolerated only for values less than 8 (as there is no ambiguity between the decimal and octal interpretations of such strings).
2. An integer that fits into 32 bits.
3. An integer packed into a bytes object of length 4 (most significant octet first).

```python
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```
**version**

The appropriate version number: 4 for IPv4, 6 for IPv6.

**max_prefixlen**

The total number of bits in the address representation for this version: 32 for IPv4, 128 for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

**compressed**

**exploded**

The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as \texttt{str(addr)} for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

**packed**

The binary representation of this address - a \texttt{bytes} object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

**reverse_pointer**

The name of the reverse DNS PTR record for the IP address, e.g.:

```python
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer['1.0.0.127.in-addr.arpa']
```
ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

*New in version 3.5.*

**is_multicast**
True if the address is reserved for multicast use. See [RFC 3171](https://tools.ietf.org/html/rfc3171) (for IPv4) or [RFC 2373](https://tools.ietf.org/html/rfc2373) (for IPv6).

**is_private**
True if the address is allocated for private networks. See [iana-ipv4-special-registry](https://www.iana.org/assignments/ipv4-special-registry) (for IPv4) or [iana-ipv6-special-registry](https://www.iana.org/assignments/ipv6-special-registry) (for IPv6).

**is_global**
True if the address is allocated for public networks. See [iana-ipv4-special-registry](https://www.iana.org/assignments/ipv4-special-registry) (for IPv4) or [iana-ipv6-special-registry](https://www.iana.org/assignments/ipv6-special-registry) (for IPv6).

*New in version 3.4.*

**is_unspecified**

**is_reserved**
True if the address is otherwise IETF reserved.

**is_loopback**
True if this is a loopback address. See [RFC 3330](https://tools.ietf.org/html/rfc3330) (for IPv4) or
**RFC 2373** (for IPv6).

**is_link_local**
True if the address is reserved for link-local usage. See **RFC 3927**.

```python
class ipaddress.IPv6Address(address)
```
Construct an IPv6 address. An **AddressValueError** is raised if `address` is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, "0000:0000:0000:0000:0000:0abc:0007:0def" can be compressed to "::abc:7:def".
2. An integer that fits into 128 bits.
3. An integer packed into a **bytes** object of length 16, big-endian.

```python
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

**compressed**

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by **str(addr)** for IPv6 addresses.
**exploded**

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes, see the corresponding documentation of the `IPv4Address` class:

- **packed**
- **reverse_pointer**
- **version**
- **max_prefixlen**
- **is_multicast**
- **is_private**
- **is_global**
- **is_unspecified**
- **is_reserved**
- **is_loopback**
- **is_link_local**
  
  *New in version 3.4: is_global*

**is_site_local**

True if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by RFC 3879. Use `is_private` to test if this address is in the space of unique local addresses as defined by RFC 4193.

**ipv4_mapped**
For addresses that appear to be IPv4 mapped addresses (starting with ::FFFF/96), this property will report the embedded IPv4 address. For any other address, this property will be None.

**sixtofour**

For addresses that appear to be 6to4 addresses (starting with 2002::/16) as defined by RFC 3056, this property will report the embedded IPv4 address. For any other address, this property will be None.

**teredo**

For addresses that appear to be Teredo addresses (starting with 2001::/32) as defined by RFC 4380, this property will report the embedded (server, client) IP address pair. For any other address, this property will be None.

**Conversion to Strings and Integers**

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the `str()` and `int()` built-in functions:

```python
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```
Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples:

```python
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```python
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```
IP Network definitions

The IPv4Network and IPv6Network objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a mask and a network address, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask 255.255.255.0 and the network address 192.168.1.0 consists of IP addresses in the inclusive range 192.168.1.0 to 192.168.1.255.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A prefix /<nbhits> is a notation that denotes how many high-order bits are set in the network mask. A net mask is an IP address with some number of high-order bits set. Thus the prefix /24 is equivalent to the net mask 255.255.255.0 in IPv4, or ffff:ff00:: in IPv6. In addition, a host mask is the logical inverse of a net mask, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to /24 in IPv4 is 0.0.0.255.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between IPv4Network and IPv6Network, so to avoid duplication they are only documented for IPv4Network. Network objects are hashable, so they can be
used as keys in dictionaries.

class ipaddress.**IPv4Network**(address, strict=True)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (/). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be /32.

   For example, the following *address* specifications are equivalent:
   
   192.168.1.0/24, 192.168.1.0/255.255.255.0 and 192.168.1.0/0.0.0.255.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being /32.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. 24) or a string representing the prefix mask (e.g.
An `AddressValueError` is raised if `address` is not a valid IPv4 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv4 address.

If `strict` is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise `TypeError` if the argument's IP version is incompatible to `self`.

*Changed in version 3.5:* Added the two-tuple form for the `address` constructor parameter.

- `version`
- `max_prefixlen`
  - Refer to the corresponding attribute documentation in `IPv4Address`.
- `is_multicast`
- `is_private`
- `is_unspecified`
- `is_reserved`
- `is_loopback`
- `is_link_local`
  - These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.
**network_address**

The network address for the network. The network address and the prefix length together uniquely define a network.

**broadcast_address**

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

**hostmask**

The host mask, as an `IPv4Address` object.

**netmask**

The net mask, as an `IPv4Address` object.

**with_prefixlen**

**compressed**

**exploded**

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

**with_netmask**

A string representation of the network, with the mask in net mask notation.

**with_hostmask**

A string representation of the network, with the mask in host mask notation.
**num_addresses**

The total number of addresses in the network.

**prefixlen**

Length of the network prefix, in bits.

**hosts()**

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```python
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'), IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'), IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
```

**overlaps(other)**

`True` if this network is partly or wholly contained in `other` or `other` is wholly contained in this network.

**address_exclude(network)**

Computes the network definitions resulting from removing the given `network` from this one. Returns an iterator of network objects. Raises `ValueError` if `network` is not completely contained in this network.

```python
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
```
subnets(prefixlen_diff=1, new_prefix=None)

The subnets that join to make the current network definition, depending on the argument values. prefixlen_diff is the amount our prefix length should be increased by. new_prefix is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of prefixlen_diff and new_prefix must be set. Returns an iterator of network objects.

supernet(prefixlen_diff=1, new_prefix=None)

The supernet containing this network definition, depending on the argument values. prefixlen_diff is the amount our prefix length should be decreased by. new_prefix is the desired new prefix of the supernet; it must be smaller than our prefix. One
and only one of `prefixlen_diff` and `new_prefix` must be set. Returns a single network object.

```python
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix)
IPv4Network('192.0.0.0/20')
```

### subnet_of(other)
Returns `True` if this network is a subnet of `other`.

```python
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> bsubnet_of(a)
True
```

*New in version 3.7.*

### supernet_of(other)
Returns `True` if this network is a supernet of `other`.

```python
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> asubnet_of(b)
True
```

*New in version 3.7.*

### compare_networks(other)
Compare this network to `other`. In this comparison only the network addresses are considered; host bits aren’t. Returns either `-1`, `0` or `1`.

```python
```
class ipaddress.IPv6Network(address, strict=True)

Construct an IPv6 network definition. address can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (/). The IP address is the network address, and the prefix length must be a single number, the prefix. If no prefix length is provided, it's considered to be /128.

   Note that currently expanded netmasks are not supported. That means 2001:db00::0/24 is a valid argument while 2001:db00::0/ffff:ff00:: not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being address and the mask being /128.

3. An integer packed into a bytes object of length 16, big-endian. The interpretation is similar to an integer address.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a

Deprecation since version 3.7: It uses the same ordering and comparison algorithm as "<", "==", and ">"
16-bytes packed integer, or an existing IPv6Address object; and the netmask is an integer representing the prefix length.

An `AddressValueError` is raised if `address` is not a valid IPv6 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv6 address.

If `strict` is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

*Changed in version 3.5:* Added the two-tuple form for the `address` constructor parameter.

```
version
max_prefixlen
is_multicast
is_private
is_unspecified
is_reserved
is_loopback
is_link_local
network_address
broadcast_address
hostmask
netmask
with_prefixlen
```
compressed
exploded
with_netmask
with_hostmask
num_addresses
prefixlen
hosts()
  Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

overlaps(other)

address_exclude(network)

subnets(prefixlen_diff=1, new_prefix=None)

supernet(prefixlen_diff=1, new_prefix=None)

subnet_of(other)

supernet_of(other)

compare_networks(other)
  Refer to the corresponding attribute documentation in IPv4Network.

is_site_local
  These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.
Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, all hosts are returned, including unusable hosts (for usable hosts, use the hosts() method). An example:

```python
>>> for addr in IPv4Network('192.0.2.0/28):
...    addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
```
Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```python
given 192.0.2.0/28
```

```python
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
```

```python
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
```

```python
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
```

```python
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```
Interface objects

Interface objects are hashable, so they can be used as keys in dictionaries.

```
class ipaddress.IPv4Interface(address)
    Construct an IPv4 interface. The meaning of address is as in the
    constructor of IPv4Network, except that arbitrary host addresses
    are always accepted.

    IPv4Interface is a subclass of IPv4Address, so it inherits all
    the attributes from that class. In addition, the following attributes
    are available:

    ip
        The address (IPv4Address) without network information.

        >>> interface = IPv4Interface('192.0.2.5/24')
        >>> interface.ip
        IPv4Address('192.0.2.5')

    network
        The network (IPv4Network) this interface belongs to.

        >>> interface = IPv4Interface('192.0.2.5/24')
        >>> interface.network
        IPv4Network('192.0.2.0/24')

    with_prefixlen
        A string representation of the interface with the mask in prefix
        notation.
```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'

**with_netmask**

A string representation of the interface with the network as a net mask.

>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'

**with_hostmask**

A string representation of the interface with the network as a host mask.

>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'

**class ipaddress.IPv6Interface(address)**

Construct an IPv6 interface. The meaning of address is as in the constructor of IPv6Network, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of IPv6Address, so it inherits all the attributes from that class. In addition, the following attributes are available:

- **ip**
- **network**
- **with_prefixlen**
**with_netmask**

**with_hostmask**

Refer to the corresponding attribute documentation in `IPv4Interface`.

**Operators**

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

**Logical operators**

Interface objects can be compared with the usual set of logical operators.

For equality comparison (`==` and `!=`), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (`<`, `>`, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.
Other Module Level Functions

The module also provides the following module level functions:

### `ipaddress.v4_int_to_packed(address)`

Represent an address as 4 packed bytes in network (big-endian) order. `address` is an integer representation of an IPv4 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv4 IP address.

```python
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

### `ipaddress.v6_int_to_packed(address)`

Represent an address as 16 packed bytes in network (big-endian) order. `address` is an integer representation of an IPv6 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv6 IP address.

### `ipaddress.summarize_address_range(first, last)`

Return an iterator of the summarized network range given the first and last IP addresses. `first` is the first `IPv4Address` or `IPv6Address` in the range and `last` is the last `IPv4Address` or `IPv6Address` in the range. A `TypeError` is raised if `first` or `last` are not IP addresses or are not of the same version. A `ValueError` is raised if `last` is not greater than `first` or if `first` address version is not 4 or 6.

```python
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(ipaddress.IPv4Address('192.0.2.0'), ...         ipaddress.IPv4Address('192.0.2.255'))]
```
ipaddress. **collapse_addresses**(addresses)

Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A **TypeError** is raised if *addresses* contains mixed version objects.

```python
>>> [ipaddr for ipaddr in ... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.130/32')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress. **get_mixed_type_key**(obj)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```python
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the key argument to *sorted()*.

*obj* is either a network or address object.
Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

```python
exception ipaddress.AddressValueError(ValueError)
    Any value error related to the address.

exception ipaddress.NetmaskValueError(ValueError)
    Any value error related to the net mask.
```
Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here’s an overview:

- **audioop** — Manipulate raw audio data
- **aifc** — Read and write AIFF and AIFC files
- **sunau** — Read and write Sun AU files
  - AU_read Objects
  - AU_write Objects
- **wave** — Read and write WAV files
  - Wave_read Objects
  - Wave_write Objects
- **chunk** — Read IFF chunked data
- **colorsyst** — Conversions between color systems
- **imghdr** — Determine the type of an image
- **sndhdr** — Determine type of sound file
- **ossaudiodev** — Access to OSS-compatible audio devices
  - Audio Device Objects
  - Mixer Device Objects
The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16, 24 or 32 bits wide, stored in bytes-like objects. All scalar items are integers, unless specified otherwise.

*Changed in version 3.4:* Support for 24-bit samples was added. All functions now accept any bytes-like object. String input now results in an immediate error.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

**exception** `audioop.error`

This exception is raised on all errors, such as unknown number of bytes per sample, etc.

`audioop.add(fragment1, fragment2, width)`

Return a fragment which is the addition of the two samples passed as parameters. `width` is the sample width in bytes, either 1, 2, 3 or 4. Both fragments should have the same length. Samples are truncated in case of overflow.
audioop.**adpcm2lin**(adpcmfragment, width, state)
Decode an Intel/DVI ADPCM coded fragment to a linear fragment.
See the description of [lin2adpcm()](#) for details on ADPCM coding.
Return a tuple (sample, newstate) where the sample has the width specified in width.

audioop.**alaw2lin**(fragment, width)
Convert sound fragments in a-LAW encoding to linearly encoded sound fragments.
a-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

audioop.**avg**(fragment, width)
Return the average over all samples in the fragment.

audioop.**avgpp**(fragment, width)
Return the average peak-peak value over all samples in the fragment.
No filtering is done, so the usefulness of this routine is questionable.

audioop.**bias**(fragment, width, bias)
Return a fragment that is the original fragment with a bias added to each sample.
Samples wrap around in case of overflow.

audioop.**byteswap**(fragment, width)
“Byteswap” all samples in a fragment and returns the modified fragment.
Converts big-endian samples to little-endian and vice versa.

*New in version 3.4.*

audioop.**cross**(fragment, width)
Return the number of zero crossings in the fragment passed as an argument.
audioop.\texttt{findfactor}(\texttt{fragment, reference})

Return a factor $F$ such that $\text{rms}(\text{add(}\text{fragment, mul(}\text{reference, -F}))$ is minimal, i.e., return the factor with which you should multiply $\text{reference}$ to make it match as well as possible to $\text{fragment}$. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to $\text{len(fragment)}$.

audioop.\texttt{findfit}(\texttt{fragment, reference})

Try to match $\text{reference}$ as well as possible to a portion of $\text{fragment}$ (which should be the longer fragment). This is (conceptually) done by taking slices out of $\text{fragment}$, using $\text{findfactor()}$ to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple $(\text{offset, factor})$ where $\text{offset}$ is the (integer) offset into $\text{fragment}$ where the optimal match started and $\text{factor}$ is the (floating-point) factor as per $\text{findfactor()}$.

audioop.\texttt{findmax}(\texttt{fragment, length})

Search $\text{fragment}$ for a slice of length $\text{length}$ samples (not bytes!) with maximum energy, i.e., return $i$ for which $\text{rms(fragment[i*2:(i+length)*2])}$ is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to $\text{len(fragment)}$.

audioop.\texttt{getsample}(\texttt{fragment, width, index})

Return the value of sample $\text{index}$ from the fragment.

audioop.\texttt{lin2adpcm}(\texttt{fragment, width, state})

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM
coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple \((\text{adpcmfrag}, \text{newstate})\), and the newstate should be passed to the next call of \texttt{lin2adpcm()}. In the initial call, \texttt{None} can be passed as the state. \texttt{adpcmfrag} is the ADPCM coded fragment packed 2 4-bit values per byte.

\texttt{audioop.lin2alaw(fragment, width)}

Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

\texttt{audioop.lin2lin(fragment, width, newwidth)}

Convert samples between 1-, 2-, 3- and 4-byte formats.

\textbf{Note:} In some audio formats, such as .WAV files, 16, 24 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

\begin{verbatim}
new_frames = audioop.lin2lin(frames, old_width, new_frames = audioop.bias(new_frames, 1, 128)
\end{verbatim}

The same, in reverse, has to be applied when converting from 8 to 16, 24 or 32 bit width samples.

\texttt{audioop.lin2ulaw(fragment, width)}
Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

```
audioop.max(fragment, width)
```
Return the maximum of the *absolute value* of all samples in a fragment.

```
audioop.maxpp(fragment, width)
```
Return the maximum peak-peak value in the sound fragment.

```
audioop.minmax(fragment, width)
```
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

```
audioop.mul(fragment, width, factor)
```
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

```
audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])
```
Convert the frame rate of the input fragment.

*state* is a tuple containing the state of the converter. The converter returns a tuple *(newfragment, newstate)*, and *newstate* should be passed to the next call of *ratecv()*.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to 1 and 0 respectively.
audioop.\texttt{reverse}(\texttt{fragment}, \texttt{width})

Reverse the samples in a fragment and returns the modified fragment.

audioop.\texttt{rms}(\texttt{fragment}, \texttt{width})

Return the root-mean-square of the fragment, i.e. \( \sqrt{\frac{\text{sum}(S_i^2)}{n}} \).

This is a measure of the power in an audio signal.

audioop.\texttt{tomono}(\texttt{fragment}, \texttt{width}, \texttt{lfactor}, \texttt{rfactor})

Convert a stereo fragment to a mono fragment. The left channel is multiplied by \texttt{lfactor} and the right channel by \texttt{rfactor} before adding the two channels to give a mono signal.

audioop.\texttt{tostereo}(\texttt{fragment}, \texttt{width}, \texttt{lfactor}, \texttt{rfactor})

Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by \texttt{lfactor} and right channel samples by \texttt{rfactor}.

audioop.\texttt{ulaw2lin}(\texttt{fragment}, \texttt{width})

Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so \texttt{width} refers only to the sample width of the output fragment here.

Note that operations such as \texttt{mul()} or \texttt{max()} make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

\begin{verbatim}
def mul_stereo(sample, width, lfactor, rfactor):
\end{verbatim}
If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the initial state (the one you passed to `lin2adpcm()`) along to the decoder, not the final state (as returned by the coder). If you want to use `struct.Struct` to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```python
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)  # one
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test
```
# Optional (for better cancellation):
# factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)),

prefill = '\0'*(pos+ipos)*2
postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
outputdata = prefill + audioop.mul(outputdata, 2)

return audioop.add(inputdata, outputdata, 2)
This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of \texttt{nchannels} \times \texttt{samplesize} bytes, and a second's worth of audio consists of \texttt{nchannels} \times \texttt{samplesize} \times \texttt{framerate} bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module \texttt{aifc} defines the following function:

\texttt{aifc.open(file, mode=None)}

Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument \texttt{file} is either a string naming a file or a file object. \texttt{mode} must be \texttt{'r'} or \texttt{'rb'}. 
when the file must be opened for reading, or 'w' or 'wb' when
the file must be opened for writing. If omitted, file.mode is used
if it exists, otherwise 'rb' is used. When used for writing, the file
object should be seekable, unless you know ahead of time how
many samples you are going to write in total and use
writeframesraw() and setnframes(). The open() function
may be used in a with statement. When the with block
completes, the close() method is called.

*Changed in version 3.4:* Support for the with statement was
added.

Objects returned by open() when a file is opened for reading have
the following methods:

*aifc.*getnchannels()
  Return the number of audio channels (1 for mono, 2 for stereo).

*aifc.*getsampwidth()
  Return the size in bytes of individual samples.

*aifc.*getframerate()
  Return the sampling rate (number of audio frames per second).

*aifc.*getnframes()
  Return the number of audio frames in the file.

*aifc.*getcomptype()
  Return a bytes array of length 4 describing the type of compression
  used in the audio file. For AIFF files, the returned value is
  b'NONE'.

*aifc.*getcompname()
Return a bytes array convertible to a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is b'not compressed'.

**aifc.getparams()**

Returns a namedtuple(\(n\)channels, \(\text{sampwidth}\), \(\text{framerate}\), \(\text{nframes}\), \(\text{comptype}\), \(\text{compname}\)), equivalent to output of the get*() methods.

**aifc.getmarkers()**

Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

**aifc.getmark(id)**

Return the tuple as described in getmarkers() for the mark with the given \(id\).

**aifc.readframes(nframes)**

Read and return the next \(n\)frames frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

**aifc.rewind()**

Rewind the read pointer. The next readframes() will start from the beginning.

**aifc.setpos(pos)**

Seek to the specified frame number.

**aifc.tell()**
Return the current frame number.

\texttt{aifc.close()}

Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by \texttt{open()} when a file is opened for writing have all the above methods, except for \texttt{readframes()} and \texttt{setpos()}. In addition the following methods exist. The \texttt{get*()} methods can only be called after the corresponding \texttt{set*()} methods have been called. Before the first \texttt{writeframes()} or \texttt{writeframesraw()}, all parameters except for the number of frames must be filled in.

\texttt{aifc.aiff()}

Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in ".aiff" in which case the default is an AIFF file.

\texttt{aifc.aifc()}

Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in ".aiff" in which case the default is an AIFF file.

\texttt{aifc.setnchannels}(nchannels)

Specify the number of channels in the audio file.

\texttt{aifc.setsampwidth}(width)

Specify the size in bytes of audio samples.

\texttt{aifc.setframerate}(rate)

Specify the sampling frequency in frames per second.

\texttt{aifc.setnframes}(nframes)
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

\texttt{aifc.setcomptype(type, name)}

Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type as a bytes array, the type parameter should be a bytes array of length 4. Currently the following compression types are supported: \texttt{b'NONE'}, \texttt{b'ULAW'}, \texttt{b'ALAW'}, \texttt{b'G722'}.

\texttt{aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)}

Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a \texttt{getparams()} call as argument to \texttt{setparams()}. 

\texttt{aifc.setmark(id, pos, name)}

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before \texttt{close()}. 

\texttt{aifc.tell()}

Return the current write position in the output file. Useful in combination with \texttt{setmark()}. 

\texttt{aifc.writeframes(data)}

Write data to the output file. This method can only be called after the audio file parameters have been set. 

*Changed in version 3.4:* Any bytes-like object is now accepted.
aifc.writeframesraw(data)

Like writeframes(), except that the header of the audio file is not updated.

*Changed in version 3.4:* Any bytes-like object is now accepted.

aifc.close()

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.
sunau — Read and write Sun AU files

Source code: Lib/sunau.py

The sunau module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules aifc and wave.

An audio file consists of a header followed by the data. The fields of the header are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic word</td>
<td>The four bytes .snd.</td>
</tr>
<tr>
<td>header size</td>
<td>Size of the header, including info, in bytes.</td>
</tr>
<tr>
<td>data size</td>
<td>Physical size of the data, in bytes.</td>
</tr>
<tr>
<td>encoding</td>
<td>Indicates how the audio samples are encoded.</td>
</tr>
<tr>
<td>sample rate</td>
<td>The sampling rate.</td>
</tr>
<tr>
<td># of channels</td>
<td>The number of channels in the samples.</td>
</tr>
<tr>
<td>info</td>
<td>ASCII string giving a description of the audio file (padded with null bytes).</td>
</tr>
</tbody>
</table>

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The sunau module defines the following functions:

sunau.open(file, mode)
If `file` is a string, open the file by that name, otherwise treat it as a seekable file-like object. `mode` can be any of

- `'r'`
  Read only mode.
- `'w'`
  Write only mode.

Note that it does not allow read/write files.

A `mode` of `'r'` returns an AU_read object, while a `mode` of `'w'` or `'wb'` returns an AU_write object.

```
sunau.openfp(file, mode)
```

A synonym for `open()`, maintained for backwards compatibility.

*Deprecated since version 3.7, will be removed in version 3.9.*

The `sunau` module defines the following exception:

```
exception sunau.Error
```

An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The `sunau` module defines the following data items:

- `sunau.AUDIO_FILE_MAGIC`
  An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

- `sunau.AUDIO_FILE_ENCODING_MULAW_8`
- `sunau.AUDIO_FILE_ENCODING_LINEAR_8`
- `sunau.AUDIO_FILE_ENCODING_LINEAR_16`
- `sunau.AUDIO_FILE_ENCODING_LINEAR_24`
Values of the encoding field from the AU header which are supported by this module.

Additional known values of the encoding field from the AU header, but which are not supported by this module.
AU_read Objects

AU_read objects, as returned by open() above, have the following methods:

AU_read.close()
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.getnchannels()
Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.getsampwidth()
Returns sample width in bytes.

AU_read.getframerate()
Returns sampling frequency.

AU_read.getnframes()
Returns number of audio frames.

AU_read.getcomptype()
Returns compression type. Supported compression types are 'ULAW', 'ALAW' and 'NONE'.

AU_read.getcompname()
Human-readable version of getcomptype(). The supported types have the respective names 'CCITT G.711 u-law', 'CCITT G.711 A-law' and 'not compressed'.

AU_read.getparams()
Returns a namedtuple(nchannels, sampwidth,
framerate, nframes, comptype, compname), equivalent to output of the get*() methods.

**AU_read.readframes(n)**
Reads and returns at most \( n \) frames of audio, as a bytes object. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

**AU_read.rewind()**
Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

**AU_read.setpos(pos)**
Set the file pointer to the specified position. Only values returned from `tell()` should be used for `pos`.

**AU_read.tell()**
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the aifc, and don’t do anything interesting.

**AU_read.getmarkers()**
Returns None.

**AU_read.getmark(id)**
Raise an error.
AU_write Objects

AU_write objects, as returned by `open()` above, have the following methods:

**AU_write.setnchannels(n)**
Set the number of channels.

**AU_write.setsampwidth(n)**
Set the sample width (in bytes.)

*Changed in version 3.4:* Added support for 24-bit samples.

**AU_write.setframerate(n)**
Set the frame rate.

**AU_write.setnframes(n)**
Set the number of frames. This can be later changed, when and if more frames are written.

**AU_write.setcomptype(type, name)**
Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

**AU_write.setparams(tuple)**
The tuple should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the set*() methods. Set all parameters.

**AU_write.tell()**
Return current position in the file, with the same disclaimer for the `AU_read.tell()` and `AU_read.setpos()` methods.
AU_write.writeframesraw(data)
Write audio frames, without correcting nframes.

*Changed in version 3.4:* Any bytes-like object is now accepted.

AU_write.writeframes(data)
Write audio frames and make sure nframes is correct.

*Changed in version 3.4:* Any bytes-like object is now accepted.

AU_write.close()
Make sure nframes is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw().
wave — Read and write WAV files

Source code: Lib/wave.py

The wave module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The wave module defines the following function and exception:

```
wave.open(file, mode=None)
```

If file is a string, open the file by that name, otherwise treat it as a file-like object. mode can be:

-'rb'
  Read only mode.

-'wb'
  Write only mode.

Note that it does not allow read/write WAV files.

A mode of 'rb' returns a Wave_read object, while a mode of 'wb' returns a Wave_write object. If mode is omitted and a file-like object is passed as file, file.mode is used as the default value for mode.

If you pass in a file-like object, the wave object will not close it when its close() method is called; it is the caller’s responsibility to close the file object.

The open() function may be used in a with statement. When the
with block completes, the `Wave_read.close()` or `Wave_write.close()` method is called.

*Changed in version 3.4:* Added support for unseekable files.

```python
wave.openfp(file, mode)
```

A synonym for `open()`, maintained for backwards compatibility.

*Deprecated since version 3.7, will be removed in version 3.9.*

```python
exception wave.Error
```

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.
Wave_read Objects

Wave_read objects, as returned by `open()`, have the following methods:

Wave_read.close()
Close the stream if it was opened by `wave`, and make the instance unusable. This is called automatically on object collection.

Wave_read.getnchannels()
Returns number of audio channels (`1` for mono, `2` for stereo).

Wave_read.getsampwidth()
Returns sample width in bytes.

Wave_read.getframerate()
Returns sampling frequency.

Wave_read.getnframes()
Returns number of audio frames.

Wave_read.getcomptype()
Returns compression type ("NONE" is the only supported type).

Wave_read.getcompname()
Human-readable version of `getcomptype()`. Usually 'not compressed' parallels 'NONE'.

Wave_read.getparams()
Returns a `namedtuple()` (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalent to output of the get*() methods.
Wave_read. **readframes**(\(n\))
Reads and returns at most \(n\) frames of audio, as a `bytes` object.

Wave_read. **rewind**()
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the `aifc` module, and don’t do anything interesting.

Wave_read. **getmarkers**()
Returns None.

Wave_read. **getmark**(\(id\))
Raise an error.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

Wave_read. **setpos**(\(pos\))
Set the file pointer to the specified position.

Wave_read. **tell**()
Return current file pointer position.
Wave_write Objects

For seekable output streams, the wave header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the nframes value must be accurate when the first frame data is written. An accurate nframes value can be achieved either by calling setnframes() or setparams() with the number of frames that will be written before close() is called and then using writeframesraw() to write the frame data, or by calling writeframes() with all of the frame data to be written. In the latter case writeframes() will calculate the number of frames in the data and set nframes accordingly before writing the frame data.

Wave_write objects, as returned by open(), have the following methods:

*Changed in version 3.4:* Added support for unseekable files.

Wave_write.close()
Make sure nframes is correct, and close the file if it was opened by wave. This method is called upon object collection. It will raise an exception if the output stream is not seekable and nframes does not match the number of frames actually written.

Wave_write.setnchannels(n)
Set the number of channels.

Wave_write.setsampwidth(n)
Set the sample width to n bytes.

Wave_write.setframerate(n)
Set the frame rate to n.
Changed in version 3.2: A non-integral input to this method is rounded to the nearest integer.

Wave_write.setnframes(n)
Set the number of frames to \( n \). This will be changed later if the number of frames actually written is different (this update attempt will raise an error if the output stream is not seekable).

Wave_write.setcomptype(type, name)
Set the compression type and description. At the moment, only compression type NONE is supported, meaning no compression.

Wave_write.setparams(tuple)
The tuple should be \((nchannels, sampwidth, framerate, nframes, comptype, compname)\), with values valid for the set*() methods. Sets all parameters.

Wave_write.tell()
Return current position in the file, with the same disclaimer for the Wave_read.tell() and Wave_read.setpos() methods.

Wave_write.writeframesraw(data)
Write audio frames, without correcting \( nframes \).

Changed in version 3.4: Any bytes-like object is now accepted.

Wave_write.writeframes(data)
Write audio frames and make sure \( nframes \) is correct. It will raise an error if the output stream is not seekable and the total number of frames that have been written after \( data \) has been written does not match the previously set value for \( nframes \).

Changed in version 3.4: Any bytes-like object is now accepted.
Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`, and any attempt to do so will raise `wave.Error`. 
chunk — Read IFF chunked data

Source code: Lib/chunk.py

This module provides an interface for reading files that use EA IFF 85 chunks. [1] This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Chunk ID</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Size of chunk in big-endian byte order, not including the header</td>
</tr>
<tr>
<td>8</td>
<td>$n$</td>
<td>Data bytes, where $n$ is the size given in the preceding field</td>
</tr>
<tr>
<td>$8 + n$</td>
<td>0 or 1</td>
<td>Pad byte needed if $n$ is odd and chunk alignment is used</td>
</tr>
</tbody>
</table>

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the Chunk class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the
file, creating a new instance will fail with an \texttt{EOFError} exception.

\begin{verbatim}
class chunk. \textbf{Chunk(file, align=True, bigendian=True, inclheader=False)}

Class which represents a chunk. The \texttt{file} argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is \texttt{read()}. If the methods \texttt{seek()} and \texttt{tell()} are present and don’t raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument \texttt{align} is true, chunks are assumed to be aligned on 2-byte boundaries. If \texttt{align} is false, no alignment is assumed. The default value is true. If the optional argument \texttt{bigendian} is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument \texttt{inclheader} is true, the size given in the chunk header includes the size of the header. The default value is false.

A \textbf{Chunk} object supports the following methods:

\begin{verbatim}
\textbf{getname()}
\end{verbatim}

Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

\begin{verbatim}
\textbf{getsize()}
\end{verbatim}

Returns the size of the chunk.

\begin{verbatim}
\textbf{close()}
\end{verbatim}

Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise \texttt{OSError} if called after the \texttt{close()} method has been called. Before Python 3.3, they used to
raise IOError, now an alias of OSError.

**isatty()**

Returns False.

**seek(pos, whence=0)**

Set the chunk's current position. The *whence* argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

**tell()**

Return the current position into the chunk.

**read(size=-1)**

Read at most *size* bytes from the chunk (less if the read hits the end of the chunk before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until the end of the chunk. An empty bytes object is returned when the end of the chunk is encountered immediately.

**skip()**

Skip to the end of the chunk. All further calls to `read()` for the chunk will return b''. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

**Footnotes**

"EA IFF 85" Standard for Interchange Format Files, Jerry
The `colors` module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

The `colors` module defines the following functions:

- `rgb_to_yiq(r, g, b)`
  - Convert the color from RGB coordinates to YIQ coordinates.

- `yiq_to_rgb(y, i, q)`
  - Convert the color from YIQ coordinates to RGB coordinates.

- `rgb_to_hls(r, g, b)`
  - Convert the color from RGB coordinates to HLS coordinates.

- `hls_to_rgb(h, l, s)`

See also: More information about color spaces can be found at [http://poynton.ca/ColorFAQ.html](http://poynton.ca/ColorFAQ.html) and [https://www.cambridgeincolour.com/tutorials/color-spaces.htm](https://www.cambridgeincolour.com/tutorials/color-spaces.htm).
Convert the color from HLS coordinates to RGB coordinates.

colorsys.\texttt{rgb\_to\_hsv}(r, g, b)

Convert the color from RGB coordinates to HSV coordinates.

colorsys.\texttt{hsv\_to\_rgb}(h, s, v)

Convert the color from HSV coordinates to RGB coordinates.

Example:

```python
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```
Multimedia Services
**imghdr — Determine the type of an image**

**Source code:** Lib/imghdr.py

The *imghdr* module determines the type of image contained in a file or byte stream.

The *imghdr* module defines the following function:

```
imghdr.what(filename, h=None)
```

Tests the image data contained in the file named by `filename`, and returns a string describing the image type. If optional `h` is provided, the `filename` is ignored and `h` is assumed to contain the byte stream to test.

*Changed in version 3.6:* Accepts a path-like object.

The following image types are recognized, as listed below with the return value from `what()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image format</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rgb'</td>
<td>SGI ImgLib Files</td>
</tr>
<tr>
<td>'gif'</td>
<td>GIF 87a and 89a Files</td>
</tr>
<tr>
<td>'pbm'</td>
<td>Portable Bitmap Files</td>
</tr>
<tr>
<td>'pgm'</td>
<td>Portable Graymap Files</td>
</tr>
<tr>
<td>'ppm'</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>'tiff'</td>
<td>TIFF Files</td>
</tr>
<tr>
<td>File Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>'rast'</td>
<td>Sun Raster Files</td>
</tr>
<tr>
<td>'xbm'</td>
<td>X Bitmap Files</td>
</tr>
<tr>
<td>'jpeg'</td>
<td>JPEG data in JFIF or Exif formats</td>
</tr>
<tr>
<td>'bmp'</td>
<td>BMP files</td>
</tr>
<tr>
<td>'png'</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>'webp'</td>
<td>WebP files</td>
</tr>
<tr>
<td>'exr'</td>
<td>OpenEXR Files</td>
</tr>
</tbody>
</table>

New in version 3.5: The `exr` and `webp` formats were added.

You can extend the list of file types `imghdr` can recognize by appending to this variable:

`imghdr.tests`

A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Example:

```python
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```
sndhdr — Determine type of sound file

Source code: Lib/sndhdr.py

The sndhdr provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a namedtuple(), containing five attributes: (filetype, framerate, nchannels, nframes, sampwidth). The value for type indicates the data type and will be one of the strings 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul'. The sampling_rate will be either the actual value or 0 if unknown or difficult to decode. Similarly, channels will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for frames will be either the number of frames or -1. The last item in the tuple, bits_per_sample, will either be the sample size in bits or 'A' for A-LAW or 'U' for u-LAW.

sndhdr.what(filename)
Determines the type of sound data stored in the file filename using whathdr(). If it succeeds, returns a namedtuple as described above, otherwise None is returned.

Changed in version 3.5: Result changed from a tuple to a namedtuple.

sndhdr.whathdr(filename)
Determines the type of sound data stored in a file based on the file
header. The name of the file is given by *filename*. This function returns a namedtuple as described above on success, or `None`.

*Changed in version 3.5:* Result changed from a tuple to a namedtuple.
ossaudiodev — Access to OSS-compatible audio devices

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

*Changed in version 3.3:* Operations in this module now raise **OSError** where **IOError** was raised.

**See also:**

**Open Sound System Programmer’s Guide**

the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see `<sys/soundcard.h>` on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

**exception ossaudiodev.OSSAudioError**

This exception is raised on certain errors. The argument is a string describing what went wrong.

(If **ossaudiodev** receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises **OSError**. Errors detected directly by **ossaudiodev** result in **OSSAudioError**.)
(For backwards compatibility, the exception class is also available as ossaudiodev.error.)

```python
ossaudiodev.open(mode)
ossaudiodev.open(device, mode)
```

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

*device* is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to `/dev/dsp`.

*mode* is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older linuxaudiodev module which *ossaudiodev* supersedes.

```python
ossaudiodev.openmixer([device])
```

Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a
device to use. If not found, it falls back to /dev/mixer.
Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by `open()` define the following methods and (read-only) attributes:

```python
oss_audio_device.close()
```

Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

```python
oss_audio_device.fileno()
```

Return the file descriptor associated with the device.

```python
oss_audio_device.read(size)
```

Read `size` bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block `read()` until the entire requested amount of data is available.

```python
oss_audio_device.write(data)
```

Write a `bytes-like object` `data` to the audio device and return the
number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see writeall().

*Changed in version 3.5:* Writable bytes-like object is now accepted.

**oss_audio_device.writeall(data)**

Write a bytes-like object data to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until data has been completely written. If the device is in blocking mode (the default), this has the same effect as write(); writeall() is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

*Changed in version 3.5:* Writable bytes-like object is now accepted.

*Changed in version 3.2:* Audio device objects also support the context management protocol, i.e. they can be used in a with statement.

The following methods each map to exactly one ioctl() system call. The correspondence is obvious: for example, setfmt() corresponds to the SNDCTL_DSP_SETFMT ioctl, and sync() to SNDCTL_DSP_SYNC (this can be useful when consulting the OSS documentation). If the underlying ioctl() fails, they all raise OSError.

**oss_audio_device.nonblock()**

Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

**oss_audio_device.getfmts()**
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFMT_MU_LAW</td>
<td>a logarithmic encoding (used by Sun .au files and /dev/audio)</td>
</tr>
<tr>
<td>AFMT_A_LAW</td>
<td>a logarithmic encoding</td>
</tr>
<tr>
<td>AFMT_IMA_ADPCM</td>
<td>a 4:1 compressed format defined by the Interactive Multimedia Association</td>
</tr>
<tr>
<td>AFMT_U8</td>
<td>Unsigned, 8-bit audio</td>
</tr>
<tr>
<td>AFMT_S16_LE</td>
<td>Signed, 16-bit audio, little-endian byte order (as used by Intel processors)</td>
</tr>
<tr>
<td>AFMT_S16_BE</td>
<td>Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)</td>
</tr>
<tr>
<td>AFMT_S8</td>
<td>Signed, 8 bit audio</td>
</tr>
<tr>
<td>AFMT_U16_LE</td>
<td>Unsigned, 16-bit little-endian audio</td>
</tr>
<tr>
<td>AFMT_U16_BE</td>
<td>Unsigned, 16-bit big-endian audio</td>
</tr>
</tbody>
</table>

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device.setfmt(format)
Try to set the current audio format to format—see getfms() for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an “audio format” of AFMT_QUERY.
oss_audio_device.channnels(nchannels)
Set the number of output channels to nchannels. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.speed(samplerate)
Try to set the audio sampling rate to samplerate samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

<table>
<thead>
<tr>
<th>Rate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>default rate for /dev/audio</td>
</tr>
<tr>
<td>11025</td>
<td>speech recording</td>
</tr>
<tr>
<td>22050</td>
<td></td>
</tr>
<tr>
<td>44100</td>
<td>CD quality audio (at 16 bits/sample and 2 channels)</td>
</tr>
<tr>
<td>96000</td>
<td>DVD quality audio (at 24 bits/sample)</td>
</tr>
</tbody>
</table>

oss_audio_device.sync()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using sync().

oss_audio_device.reset()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling
reset().

oss_audio_device.post()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. format, nchannels, and samplerate should be as specified in the setfmt(), channels(), and speed() methods. If strict is true, setparameters() checks to see if each parameter was actually set to the requested value, and raises OSSAudioError if not. Returns a tuple (format, nchannels, samplerate) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of setfmt(), channels(), and speed()).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, ch
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
```
rate = dsp.rate(rate)

`oss_audio_device.bufsize()`  
Returns the size of the hardware buffer, in samples.

`oss_audio_device.obufcount()`  
Returns the number of samples that are in the hardware buffer yet to be played.

`oss_audio_device.obuffree()`  
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

`oss_audio_device.closed`  
Boolean indicating whether the device has been closed.

`oss_audio_device.name`  
String containing the name of the device file.

`oss_audio_device.mode`  
The I/O mode for the file, either "r", "rw", or "w".
Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.close()

This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an OSError.

oss_mixer_device.fileno()

Returns the file handle number of the open mixer device file.

Changed in version 3.2: Mixer objects also support the context management protocol.

The remaining methods are specific to audio mixing:

oss_mixer_device.controls()

This method returns a bitmask specifying the available mixer controls (“Control” being a specific mixable “channel”, such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```python
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIX # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that
uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

**oss_mixer_device.stereocontrols()**

Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with controls() to determine which).

See the code example for the controls() function for an example of getting data from a bitmask.

**oss_mixer_device.reccontrols()**

Returns a bitmask specifying the mixer controls that may be used to record. See the code example for controls() for an example of reading from a bitmask.

**oss_mixer_device.get(control)**

Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises **OSSAudioError** if an invalid control is specified, or **OSError** if an unsupported control is specified.

**oss_mixer_device.set(control, (left, right))**

Sets the volume for a given mixer control to (left, right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple.
Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

 Raises :class:`OSSAudioError` if an invalid mixer control was specified, or if the specified volumes were out-of-range.

 **oss_mixer_device.get_recsrc()**

 This method returns a bitmask indicating which control(s) are currently being used as a recording source.

 **oss_mixer_device.set_recsrc(bitmask)**

 Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises :class:`OSError` if an invalid source was specified. To set the current recording source to the microphone input:

 ```python
 mixer.setrecesrc (1 << ossaudiodev.SOUND_MIXER_MIC)
 ```
Internationalization

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

- **gettext** — Multilingual internationalization services
  - GNU gettext API
  - Class-based API
    - The NullTranslations class
    - The GNUTranslations class
    - Solaris message catalog support
    - The Catalog constructor
  - Internationalizing your programs and modules
    - Localizing your module
    - Localizing your application
    - Changing languages on the fly
    - Deferred translations
  - Acknowledgements
- **locale** — Internationalization services
  - Background, details, hints, tips and caveats
  - For extension writers and programs that embed Python
  - Access to message catalogs
**gettext** — Multilingual internationalization services

Source code: Lib/gettext.py

The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.
GNU gettext API

The gettext module defines the following API, which is very similar to the GNU gettext API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

```
gettext.bindtextdomain(domain, localedir=None)
```

Bind the `domain` to the locale directory `localedir`. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on Unix):

```
localedir/language/LC_MESSAGES/domain.mo,
```

where `languages` is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

If `localedir` is omitted or `None`, then the current binding for `domain` is returned. [1]

```
gettext.bind_textdomain_codeset(domain, codeset=None)
```

Bind the `domain` to `codeset`, changing the encoding of byte strings returned by the `lgettext()`, `ldgettext()`, `lngettext()` and `ldngettext()` functions. If `codeset` is omitted, then the current binding is returned.

```
gettext.textdomain(domain=None)
```

Change or query the current global domain. If `domain` is `None`, then the current global domain is returned, otherwise the global
domain is set to \textit{domain}, which is returned.

\texttt{gettext} \texttt{. \texttt{gettext}(message)}

Return the localized translation of \textit{message}, based on the current global domain, language, and locale directory. This function is usually aliased as \texttt{(_)} in the local namespace (see examples below).

\texttt{gettext} \texttt{. \texttt{dgettext}(domain, message)}

Like \texttt{gettext()}, but look the message up in the specified \textit{domain}.

\texttt{gettext} \texttt{. \texttt{ngettext}(singular, plural, n)}

Like \texttt{gettext()}, but consider plural forms. If a translation is found, apply the plural formula to \textit{n}, and return the resulting message (some languages have more than two plural forms). If no translation is found, return \textit{singular} if \textit{n} is 1; return \textit{plural} otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable \textit{n}; the expression evaluates to the index of the plural in the catalog. See the GNU \texttt{gettext documentation} for the precise syntax to be used in \texttt{.po} files and the formulas for a variety of languages.

\texttt{gettext} \texttt{. \texttt{dngettext}(domain, singular, plural, n)}

Like \texttt{ngettext()}, but look the message up in the specified \textit{domain}.

\texttt{gettext} \texttt{. \texttt{lgettext}(message)}

\texttt{gettext} \texttt{. \texttt{ldgettext}(domain, message)}

\texttt{gettext} \texttt{. \texttt{lngettext}(singular, plural, n)}
gettext. **ldgettext**(domain, singular, plural, n)

Equivalent to the corresponding functions without the `l` prefix (`gettext()`, `dgettext()`, `ngettext()` and `dngettext()`), but the translation is returned as a byte string encoded in the preferred system encoding if no other encoding was explicitly set with `bind_textdomain_codeset()`.

**Warning:** These functions should be avoided in Python 3, because they return encoded bytes. It’s much better to use alternatives which return Unicode strings instead, since most Python applications will want to manipulate human readable text as strings instead of bytes. Further, it’s possible that you may get unexpected Unicode-related exceptions if there are encoding problems with the translated strings. It is possible that the `ld*()` functions will be deprecated in future Python versions due to their inherent problems and limitations.

Note that GNU gettext also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here’s an example of typical usage for this API:

```python
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```
Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a “translations” class which implements the parsing of GNU .mo format files, and has methods for returning strings. Instances of this “translations” class can also install themselves in the built-in namespace as the function _().

gettext.find(domain, localedir=None, languages=None, all=False)

This function implements the standard .mo file search algorithm. It takes a domain, identical to what textdomain() takes. Optional localedir is as in bindtextdomain() Optional languages is a list of strings, where each string is a language code.

If localedir is not given, then the default system locale directory is used. [2] If languages is not given, then the following environment variables are searched: LANGUAGE, LC_ALL, LC_MESSAGES, and LANG. The first one returning a non-empty value is used for the languages variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

localedir/language/LC_MESSAGES/domain.mo
The first such file name that exists is returned by \texttt{find()}. If no such file is found, then \texttt{None} is returned. If \texttt{all} is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

\begin{verbatim}
gettext.translation(domain, localedir=None, languages=None, class_=None, fallback=False, codeset=None)
\end{verbatim}

Return a Translations instance based on the \texttt{domain}, \texttt{localedir}, and \texttt{languages}, which are first passed to \texttt{find()} to get a list of the associated \texttt{.mo} file paths. Instances with identical \texttt{.mo} file names are cached. The actual class instantiated is either \texttt{class_} if provided, otherwise \texttt{GNUTranslations}. The class's constructor must take a single \texttt{file object} argument. If provided, \texttt{codeset} will change the charset used to encode translated strings in the \texttt{lgettext()} and \texttt{lngettext()} methods.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, \texttt{copy.copy()} is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no \texttt{.mo} file is found, this function raises \texttt{OSError} if \texttt{fallback} is false (which is the default), and returns a \texttt{NullTranslations} instance if \texttt{fallback} is true.

\textit{Changed in version 3.3: \texttt{IOError} used to be raised instead of \texttt{OSError}.}

\begin{verbatim}
gettext.install(domain, localedir=None, codeset=None, names=None)
\end{verbatim}

This installs the function \texttt{(_)} in Python's builtins namespace, based on \texttt{domain}, \texttt{localedir}, and \texttt{codeset} which are passed to the
function translation().

For the names parameter, please see the description of the translation object's install() method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the __() function, like this:

```python
print(__('This string will be translated.'))
```

For convenience, you want the __() function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

**The NullTranslations class**

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is NullTranslations; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of NullTranslations:

```python
class gettext.NullTranslations(fp=None)
    Takes an optional file object fp, which is ignored by the base class. Initializes “protected” instance variables _info and _charset which are set by derived classes, as well as _fallback, which is set through add_fallback(). It then calls self._parse(fp) if fp is not None.

    _parse(fp)
```
No-op’d in the base class, this method takes file object \( fp \), and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

**add_fallback**(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

**gettext**(*message*)

If a fallback has been set, forward gettext() to the fallback. Otherwise, return *message*. Overridden in derived classes.

**ngettext**(*singular, plural, n*)

If a fallback has been set, forward ngettext() to the fallback. Otherwise, return *singular* if \( n \) is 1; return *plural* otherwise. Overridden in derived classes.

**lgettext**(*message*)

**lngettext**(*singular, plural, n*)

Equivalent to gettext() and ngettext(), but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with set_output_charset(). Overridden in derived classes.

**Warning:** These methods should be avoided in Python 3. See the warning for the lgettext() function.

**info**()

Return the “protected” \_info variable.
charset()

Return the encoding of the message catalog file.

output_charset()

Return the encoding used to return translated messages in lgettext() and lngettext().

set_output_charset(charset)

Change the encoding used to return translated messages.

install(names=None)

This method installs gettext() into the built-in namespace, binding it to _.

If the names parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are 'gettext', 'ngettext', 'lgettext' and 'lngettext'.

Note that this is only one way, albeit the most convenient way, to make the _() function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _(). Instead, they should use this code to make _() available to their module:

```python
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts _() only in the module’s global namespace and so only affects calls within this module.
The **GNUTranslations** class

The gettext module provides one additional class derived from NullTranslations: GNUTranslations. This class overrides _parse() to enable reading GNU gettext format .mo files in both big-endian and little-endian format.

GNUTranslations parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as the translation for the empty string. This meta-data is in RFC 822-style key: value pairs, and should contain the Project-Id-Version key. If the key Content-Type is found, then the charset property is used to initialize the “protected” _charset instance variable, defaulting to None if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII encoding is assumed.

Since message ids are read as Unicode strings too, all *gettext() methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the “protected” _info instance variable.

If the .mo file's magic number is invalid, the major version number is unexpected, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise OSError.

**class gettext.GNUTranslations**

The following methods are overridden from the base class implementation:
**gettext**(message)

Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's gettext() method. Otherwise, the *message* id is returned.

**ngettext**(singular, plural, n)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's ngettext() method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```python
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory'
    n) % {'num': n}
```

**lgettext**(message)

**lngettext**(singular, plural, n)

Equivalent to gettext() and ngettext(), but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with
set_output_charset().

**Warning:** These methods should be avoided in Python 3. See the warning for the `lgettext()` function.

**Solaris message catalog support**

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

**The Catalog constructor**

GNOME uses a version of the `gettext` module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```python
import gettext
cat = gettext.Catalog(domain, localedir)
_ = catgettext
print(_('hello world'))
```

For compatibility with this older module, the function `Catalog()` is an alias for the `translation()` function described above.

One difference between this module and Henstridge’s: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.
Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the gettext module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_(...)` — that is, a call to the function `_()`. For example:

```python
filename = 'mylog.txt'
message = _( 'writing a log message' )
fp = open( filename, 'w' )
fp.write(message)
fp.close()
```

In this example, the string 'writing a log message' is marked as a candidate for translation, while the strings 'mylog.txt' and 'w' are not.
There are a few tools to extract the strings meant for translation. The original GNU gettext only supported C or C++ source code but its extended version xgettext scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a pybabel script to extract and compile message catalogs. François Pinard's program called xpot does a similar job and is available as part of his po-utils package.

(Python also includes pure-Python versions of these programs, called pygettext.py and msgfmt.py; some Python distributions will install them for you. pygettext.py is similar to xgettext, but only understands Python source code and cannot handle other programming languages such as C or C++. pygettext.py supports a command-line interface similar to xgettext; for details on its use, run pygettext.py --help. msgfmt.py is binary compatible with GNU msgfmt. With these two programs, you may not need the GNU gettext package to internationalize your Python applications.)

xgettext, pygettext, and similar tools generate .po files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these .po files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a <language-name>.po file that's compiled into a machine-readable .mo binary catalog file using the msgfmt program. The .mo files are used by the gettext module for the actual translation processing at run-time.
How you use the `gettext` module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

**Localizing your module**

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU `gettext` API but instead the class-based API.

Let’s say your module is called “spam” and the module’s various natural language translation `.mo` files reside in `/usr/share/locale` in GNU `gettext` format. Here’s what you would put at the top of your module:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = tgettext
```

**Localizing your application**

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```python
import gettext
ggettext.install('myapplication')
```
If you need to set the locale directory, you can pass it into the `install()` function:

```python
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

**Changing languages on the fly**

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```python
import gettext
lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

**Deferred translations**

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:
Here, you want to mark the strings in the `animals` list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

```python
animals = ['mollusk',
          'albatross',
          'rat',
          'penguin',
          'python', ]

# ...
for a in animals:
    print(a)
```

This works because the dummy definition of `___` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `___` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `___` in the local namespace.
Note that the second use of `()` will not identify “a” as being translatable to the `gettext` program, because the parameter is not a string literal.

Another way to handle this is with the following example:

```python
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(N_(a))
```

In this case, you are marking translatable strings with the function `N_()`, which won’t conflict with any definition of `()`). However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. `xgettext`, `pygettext`, `pybabel extract`, and `xpot` all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.
Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

Footnotes

The default locale directory is system dependent; for example, on RedHat Linux it is /usr/share/locale, but on Solaris it is /usr/lib/locale. The gettext module does not try to support these system dependent defaults; instead its default is sys.prefix/share/locale. For this reason, it is always best to call bindtextdomain() with an explicit absolute path at the start of your application.

[1] See the footnote for bindtextdomain() above.
locale — Internationalization services

Source code: Lib/locale.py

The `locale` module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The `locale` module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The `locale` module defines the following exception and functions:

*exception* `locale.Error`

  Exception raised when the locale passed to `setlocale()` is not recognized.

`locale.setlocale(category, locale=None)`

  If `locale` is given and not `None`, `setlocale()` modifies the locale setting for the `category`. The available categories are listed in the data description below. `locale` may be a string, or an iterable of two strings (language code and encoding). If it’s an iterable, it’s converted to a locale name using the locale aliasing engine. An empty string specifies the user’s default settings. If the modification of the locale fails, the exception `Error` is raised. If successful, the new locale setting is returned.
If *locale* is omitted or `None`, the current setting for *category* is returned.

`setlocale()` is not thread-safe on most systems. Applications typically start with a call of

```python
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user’s default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

`locale.localeconv()`

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

<table>
<thead>
<tr>
<th>Category</th>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LC_NUMERIC</code></td>
<td>'decimal_point'</td>
<td>Decimal point character.</td>
</tr>
<tr>
<td></td>
<td>'grouping'</td>
<td>Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. The sequence is terminated with <code>CHAR_MAX</code> further grouping is performed. If the sequence terminates with a <code>CHAR_MAX</code> further perfor...</td>
</tr>
</tbody>
</table>
The last group size is repeatedly used.

<table>
<thead>
<tr>
<th>LC_MONETARY</th>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'thousands_sep'</td>
<td>Character used between groups.</td>
<td></td>
</tr>
<tr>
<td>'int_curr_symbol'</td>
<td>International currency symbol.</td>
<td></td>
</tr>
<tr>
<td>'currency_symbol'</td>
<td>Local currency symbol.</td>
<td></td>
</tr>
<tr>
<td>'p_cs_precedes/n_cs_precedes'</td>
<td>Whether the currency symbol precedes the value (for positive resp. negative values).</td>
<td></td>
</tr>
<tr>
<td>'p_sep_by_space/n_sep_by_space'</td>
<td>Whether the currency symbol is separated from the value (for positive resp. negative values).</td>
<td></td>
</tr>
<tr>
<td>'mon_decimal_point'</td>
<td>Decimal point used for monetary values.</td>
<td></td>
</tr>
<tr>
<td>'frac_digits'</td>
<td>Number of fractional digits used in local formatting of monetary values.</td>
<td></td>
</tr>
<tr>
<td>'int_frac_digits'</td>
<td>Number of fractional digits used in international formatting of monetary values.</td>
<td></td>
</tr>
</tbody>
</table>
| 'mon_thousands_sep' | Group separator used for monetary values.
All numeric values can be set to \texttt{CHAR\_MAX} to indicate that there is no value specified in this locale.

The possible values for \texttt{\textquotesingle p\_sign\_posn\textquotesingle} and \texttt{\textquotesingle n\_sign\_posn\textquotesingle} are given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Currency and value are surrounded by parentheses.</td>
</tr>
<tr>
<td>1</td>
<td>The sign should precede the value and currency symbol.</td>
</tr>
<tr>
<td>2</td>
<td>The sign should follow the value and currency symbol.</td>
</tr>
<tr>
<td>3</td>
<td>The sign should immediately precede the value.</td>
</tr>
<tr>
<td>4</td>
<td>The sign should immediately follow the value.</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>Nothing is specified in this locale.</td>
</tr>
</tbody>
</table>
The function sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

*Changed in version 3.7:* The function now sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale in some cases.

locale.\texttt{nl_langinfo}(\texttt{option})

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The \texttt{nl_langinfo()} function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.\texttt{CODESET}

Get a string with the name of the character encoding used in the selected locale.

locale.\texttt{D_T_FMT}

Get a string that can be used as a format string for \texttt{time.strftime()} to represent date and time in a locale-specific way.

locale.\texttt{D_FMT}

Get a string that can be used as a format string for \texttt{time.strftime()} to represent a date in a locale-specific way.
locale. **T_FMT**

Get a string that can be used as a format string for `time.strftime()` to represent a time in a locale-specific way.

locale. **T_FMT_AMPM**

Get a format string for `time.strftime()` to represent time in the am/pm format.

**DAY_1** ... **DAY_7**

Get the name of the n-th day of the week.

**Note:** This follows the US convention of **DAY_1** being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

**ABDAY_1** ... **ABDAY_7**

Get the abbreviated name of the n-th day of the week.

**MON_1** ... **MON_12**

Get the name of the n-th month.

**ABMON_1** ... **ABMON_12**

Get the abbreviated name of the n-th month.

locale. **RADIXCHAR**

Get the radix character (decimal dot, decimal comma, etc.).

locale. **THOUSEP**

Get the separator character for thousands (groups of three digits).

locale. **YESEXPR**
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

**Note:** The expression is in the syntax suitable for the `regex()` function from the C library, which might differ from the syntax used in `re`.

**locale. NOEXPR**

Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

**locale. CRNCYSTR**

Get the currency symbol, preceded by “-” if the symbol should appear before the value, “+” if the symbol should appear after the value, or “.” if the symbol should replace the radix character.

**locale. ERA**

Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the `time.strftime()` function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

**locale. ERA_D_T_FMT**

Get a format string for `time.strftime()` to represent date
and time in a locale-specific era-based way.

locale.ERA_D_FMT
    Get a format string for time.strftime() to represent a date in a locale-specific era-based way.

locale.ERA_T_FMT
    Get a format string for time.strftime() to represent a time in a locale-specific era-based way.

locale.ALT_DIGITS
    Get a representation of up to 100 values used to represent the values 0 to 99.

locale.getdefaultlocale([envvars])
    Tries to determine the default locale settings and returns them as a tuple of the form (language_code, encoding).

    According to POSIX, a program which has not called setlocale(LC_ALL, '') runs using the portable 'C' locale. Calling setlocale(LC_ALL, '') lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

    To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name 'LANG'. The GNU gettext search path contains 'LC_ALL', 'LC_CTYPE', 'LANG' and 'LANGUAGE', in that order.
Except for the code 'C', the language code corresponds to RFC 1766. *language code* and *encoding* may be *None* if their values cannot be determined.

**locale.getlocale(category=LC_CTYPE)**

Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the LC_* values except LC_ALL. It defaults to LC_CTYPE.

Except for the code 'C', the language code corresponds to RFC 1766. *language code* and *encoding* may be *None* if their values cannot be determined.

**locale.getpreferredencoding(do_setlocale=True)**

Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to *False*.

On Android or in the UTF-8 mode (-X utf8 option), always return 'UTF-8', the locale and the *do_setlocale* argument are ignored.

*Changed in version 3.7:* The function now always returns UTF-8 on Android or if the UTF-8 mode is enabled.

**locale.normalize(localename)**

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*.
normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like `setlocale()`.

```python
locale.resetlocale(category=LC_ALL)
```
Sets the locale for category to the default setting.

The default setting is determined by calling `getdefaultlocale()`. category defaults to LC_ALL.

```python
locale.strcoll(string1, string2)
```
Compares two strings according to the current LC_COLLATE setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether string1 collates before or after string2 or is equal to it.

```python
locale.strxfrm(string)
```
Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

```python
locale.format_string(format, val, grouping=False, monetary=False)
```
Formats a number val according to the current LC_NUMERIC setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If grouping is true, also takes the grouping into account.

If monetary is true, the conversion uses monetary thousands separator and grouping strings.
Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

*Changed in version 3.7:* The `monetary` keyword parameter was added.

```
locale.format(format, val, grouping=False, monetary=False)
```

Please note that this function works like `format_string()` but will only work for exactly one `%char` specifier. For example, `'%f'` and `'%(0)f'` are both valid specifiers, but `'f KiB'` is not.

For whole format strings, use `format_string()`.

*Deprecated since version 3.7:* Use `format_string()` instead.

```
locale.currency(val, symbol=True, grouping=False, international=False)
```

Formats a number `val` according to the current `LC_MONETARY` settings.

The returned string includes the currency symbol if `symbol` is true, which is the default. If `grouping` is true (which is not the default), grouping is done with the value. If `international` is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the ‘C’ locale, so you have to set a locale via `setlocale()` first.

```
locale.str(float)
```

Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

```
locale.delocalize(string)
```
Converts a string into a normalized number string, following the `LC_NUMERIC` settings.

*New in version 3.5.*

`locale.atof(string)`
Converts a string to a floating point number, following the `LC_NUMERIC` settings.

`locale.atoi(string)`
Converts a string to an integer, following the `LC_NUMERIC` conventions.

`locale.LC_CTYPE`
Locale category for the character type functions. Depending on the settings of this category, the functions of module `string` dealing with case change their behaviour.

`locale.LC_COLLATE`
Locale category for sorting strings. The functions `strcoll()` and `strxfrm()` of the `locale` module are affected.

`locale.LC_TIME`
Locale category for the formatting of time. The function `time.strftime()` follows these conventions.

`locale.LC_MONETARY`
Locale category for formatting of monetary values. The available options are available from the `localeconv()` function.

`locale.LC_MESSAGES`
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages
displayed by the operating system, like those returned by `os.strerror()` might be affected by this category.

**locale.LC_NUMERIC**
Locale category for formatting numbers. The functions `format()`, `atoi()`, `atof()` and `str()` of the `locale` module are affected by that category. All other numeric formatting operations are not affected.

**locale.LC_ALL**
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

**locale.CHAR_MAX**
This is a symbolic constant used for different values returned by `localeconv()`.

Example:

```python
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('řn', 'foo')  # compare a string
>>> locale.setlocale(locale.LC_ALL, '')  # use user
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default
>>> locale.setlocale(locale.LC_ALL, loc)  # restore
```
Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the C locale, no matter what the user's preferred locale is. There is one exception: the LC_CTYPE category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling setlocale(LC_ALL, '').

It is generally a bad idea to call setlocale() in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with time.strftime()), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-C locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: atof(), atoi(),
There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.
For extension writers and programs that embed Python

Extension modules should never call \texttt{setlocale()}, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is \texttt{C}).

When Python code uses the \texttt{locale} module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the \_locale extension module (which does all the work) from the table of built-in modules in the \texttt{config.c} file, and make sure that the \_locale module is not accessible as a shared library.
Access to message catalogs

locale.gettext(msg)
locale.dgettext(domain, msg)
locale.dcgettext(domain, msg, category)
locale.textdomain(domain)
locale.bindtextdomain(domain, dir)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the gettext module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use gettext instead. A known exception to this rule are applications that link with additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.
Program Frameworks

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

- **turtle — Turtle graphics**
  - Introduction
  - Overview of available Turtle and Screen methods
    - Turtle methods
    - Methods of TurtleScreen/Screen
  - Methods of RawTurtle/Turtle and corresponding functions
    - Turtle motion
    - Tell Turtle's state
    - Settings for measurement
    - Pen control
      - Drawing state
      - Color control
      - Filling
      - More drawing control
    - Turtle state
      - Visibility
      - Appearance
  - Using events
  - Special Turtle methods
  - Compound shapes
  - Methods of TurtleScreen/Screen and corresponding functions
    - Window control
    - Animation control
    - Using screen events
    - Input methods
- Settings and special methods
- Methods specific to Screen, not inherited from TurtleScreen
  - Public classes
  - Help and configuration
    - How to use help
    - Translation of docstrings into different languages
    - How to configure Screen and Turtles
  - `turtledemo` — Demo scripts
  - Changes since Python 2.6
  - Changes since Python 3.0
- `cmd` — Support for line-oriented command interpreters
  - Cmd Objects
  - Cmd Example
- `shlex` — Simple lexical analysis
  - shlex Objects
  - Parsing Rules
  - Improved Compatibility with Shells
turtle — Turtle graphics

Source code: Lib/turtle.py
Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The turtle module is an extended reimplemention of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the -n switch.

The turtle module provides turtle graphics primitives, in both object-

```
Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.
```

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
```
oriented and procedure-oriented ways. Because it uses `tkinter` for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The `TurtleScreen` class defines graphics windows as a playground for the drawing turtles. Its constructor needs a `tkinter.Canvas` or a `ScrolledCanvas` as argument. It should be used when `turtle` is used as part of some application.

The function `Screen()` returns a singleton object of a `TurtleScreen` subclass. This function should be used when `turtle` is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

All methods of `TurtleScreen/Screen` also exist as functions, i.e. as part of the procedure-oriented interface.

2. `RawTurtle` (alias: `RawPen`) defines Turtle objects which draw on a `TurtleScreen`. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

Derived from RawTurtle is the subclass `Turtle` (alias: `Pen`), which draws on “the” `Screen` instance which is automatically created, if not already present.

All methods of RawTurtle/Turtle also exist as functions, i.e. part of

```python
   forward(200)
   left(170)
   if abs(pos()) < 1:
       break
end_fill()
done()
```
the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes `Screen` and `Turtle`. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

**Note:** In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument `self` which is omitted here.
Overview of available Turtle and Screen methods

Turtle methods

Turtle motion
  Move and draw
  \n  \texttt{forward()} | \texttt{fd()}
  \texttt{backward()} | \texttt{bk() | back()}
  \texttt{right()} | \texttt{rt()}
  \texttt{left()} | \texttt{lt()}
  \texttt{goto()} | \texttt{setpos()} | \texttt{setposition()}
  \texttt{setx()}
  \texttt{sety()}
  \texttt{setheading()} | \texttt{seth()}
  \texttt{home()}
  \texttt{circle()}
  \texttt{dot()}
  \texttt{stamp()}
  \texttt{clearstamp()}
  \texttt{clearstamps()}
  \texttt{undo()}
  \texttt{speed()}

Tell Turtle's state

  \texttt{position()} | \texttt{pos()}
  \texttt{towards()}
  \texttt{xcor()}
  \texttt{ycor()}
  \texttt{heading()}
  \texttt{distance()}
Setting and measurement

\[ \text{degrees()} \]
\[ \text{radians()} \]

Pen control

Drawing state

\[ \text{pendown()} | \text{pd()} | \text{down()} \]
\[ \text{penup()} | \text{pu()} | \text{up()} \]
\[ \text{pensize()} | \text{width()} \]
\[ \text{pen()} \]
\[ \text{isdown()} \]

Color control

\[ \text{color()} \]
\[ \text{pencolor()} \]
\[ \text{fillcolor()} \]

Filling

\[ \text{filling()} \]
\[ \text{begin_fill()} \]
\[ \text{end_fill()} \]

More drawing control

\[ \text{reset()} \]
\[ \text{clear()} \]
\[ \text{write()} \]

Turtle state

Visibility

\[ \text{showturtle()} | \text{st()} \]
hideturtle() | ht()
isvisible()

Appearance

shape()
resizemode()
shapesize() | turtlesize()
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Using events

onclick()
onrelease()
ondrag()

Special Turtle methods

begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()
undobufferentries()

Methods of TurtleScreen/Screen

Window control
bgcolor()
bgpic()
clear() | clearscren()reset() | resetscreen()screensize()setworldcoordinates()

Animation control
delay()
tracer()
update()

Using screen events
listen()onkey() | onkeyrelease()onkeypress()onclick() | onscreenclck()ontimer()mainloop() | done()

Settings and special methods
mode()
colormode()getcanvas()getshapes()register_shape() | addshape()turtles()window_height()window_width()

Input methods
textinput()
numinput()

Methods specific to Screen

bye()
exitonclick()
setup()
title()
Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called turtle.

Turtle motion

turtle.\texttt{forward}(distance)
turtle.\texttt{fd}(distance)

| Parameters: | distance – a number (integer or float) |

Move the turtle forward by the specified distance, in the direction the turtle is headed.

```python
>>> turtle.position()
(0.00, 0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00, 0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00, 0.00)
```

turtle.\texttt{back}(distance)
turtle.\texttt{bk}(distance)
turtle.\texttt{backward}(distance)

| Parameters: | distance – a number |

Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle's heading.
turtle.**right**(angle)
turtle.**rt**(angle)

**Parameters:**

| angle | a number (integer or float) |

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```python
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**left**(angle)
turtle.**lt**(angle)

**Parameters:**

| angle | a number (integer or float) |

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```python
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

```python
>>> turtle.left(45)
>>> turtle.heading()
67.0
```
turtle.\texttt{goto}(x, y=\texttt{None})
turtle.\texttt{setpos}(x, y=\texttt{None})
turtle.\texttt{setposition}(x, y=\texttt{None})

\begin{tabular}{|c|l|}
\hline
**Parameters:** & \\
\hline
\hline
\textbullet{} \textit{x} – a number or a pair/vector of numbers & \\
\textbullet{} \textit{y} – a number or \texttt{None} & \\
\hline
\end{tabular}

If \textit{y} is \texttt{None}, \textit{x} must be a pair of coordinates or a \texttt{Vec2D} (e.g. as returned by \texttt{pos()}).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle’s orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00, 0.00)
>>> turtle.setpos(60, 30)
>>> turtle.pos()
(60.00, 30.00)
>>> turtle.setpos((20, 80))
>>> turtle.pos()
(20.00, 80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00, 0.00)
```

turtle.\texttt{setx}(x)

\begin{tabular}{|c|l|}
\hline
**Parameters:** & \textit{x} – a number (integer or float) & \\
\hline
\hline
```
>>> turtle.position()
(0.00, 240.00)
>>> turtle.setx(10)
```
turtle.sety(y)

Parameters:  
y – a number (integer or float)

Set the turtle's second coordinate to $y$, leave first coordinate unchanged.

```python
>>> turtle.position()
(10.00,240.00)
```  
```python
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```  

turtle.setheading(to_angle)  
turtle.seth(to_angle)

Parameters:  
to_angle – a number (integer or float)

Set the orientation of the turtle to $to\_angle$. Here are some common directions in degrees:

<table>
<thead>
<tr>
<th>standard mode</th>
<th>logo mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - east</td>
<td>0 - north</td>
</tr>
<tr>
<td>90 - north</td>
<td>90 - east</td>
</tr>
<tr>
<td>180 - west</td>
<td>180 - south</td>
</tr>
<tr>
<td>270 - south</td>
<td>270 - west</td>
</tr>
</tbody>
</table>

```python
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```
turtle.**home()**

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see `mode()`).

```python
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle(radius, extent=None, steps=None)**

**Parameters:**
- **radius** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)

Draw a circle with given `radius`. The center is `radius` units left of the turtle; `extent` – an angle – determines which part of the circle is drawn. If `extent` is not given, draw the entire circle. If `extent` is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if `radius` is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of `extent`.

As the circle is approximated by an inscribed regular polygon, `steps` determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```python
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
```
turtle.

```python
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00, 0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00, 240.00)
>>> turtle.heading()
180.0
```

turtle.

```python
dot(size=None, *color)
```

**Parameters:**

- `size` – an integer >= 1 (if given)
- `color` – a colorstring or a numeric color tuple

Draw a circular dot with diameter `size`, using `color`. If `size` is not given, the maximum of pensize+4 and 2*pensize is used.

```python
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle
>>> turtle.position()
(100.00, -0.00)
>>> turtle.heading()
0.0
```

turtle.

```python
stamp()
```

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling clearstamp(stamp_id).

```python
>>> turtle.color("blue")
```
turtle\texttt{.clearstamp}(\textit{stampid})

**Parameters:**

- **\textit{stampid}** – an integer, must be return value of previous \texttt{stamp()} call

Delete stamp with given \textit{stampid}.

```python
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle\texttt{.clearstamps}(\textit{n=None})

**Parameters:**

- **\textit{n}** – an integer (or None)

Delete all or first/last \textit{n} of turtle's stamps. If \textit{n} is None, delete all stamps, if \textit{n} > 0 delete first \textit{n} stamps, else if \textit{n} < 0 delete last \textit{n} stamps.

```python
>>> for i in range(8):
...   ...   turtle.stamp(); turtle.fd(30)
13
14
15
16
17
```
```python
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo()**

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```python
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(speed=None)

**Parameters:**

| speed | an integer in the range 0..10 or a speedstring (see below) |

Set the turtle’s speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- “fastest”: 0
- “fast”: 10
- “normal”: 6
- “slow”: 3
- “slowest”: 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.
Attention: \textit{speed} = 0 means that \textit{no} animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

\begin{verbatim}
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
\end{verbatim}

\textbf{Tell Turtle’s state}

\texttt{turtle.position()}
\texttt{turtle.pos()}

Return the turtle’s current location (x,y) (as a \texttt{Vec2D} vector).

\begin{verbatim}
>>> turtle.pos()
(440.00, -0.00)
\end{verbatim}

\texttt{turtle.towards(x, y=None)}

\textbf{Parameters:}
\begin{itemize}
  \item \texttt{x} – a number or a pair/vector of numbers or a turtle instance
  \item \texttt{y} – a number if \texttt{x} is a number, else \texttt{None}
\end{itemize}

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle’s start orientation which depends on the mode - “standard”/“world” or “logo”).

\begin{verbatim}
>>> turtle.goto(10, 10)
\end{verbatim}
turtle. `xcor()`
Return the turtle's x coordinate.

```python
>>> turtle.towards(0, 0)
225.0
```

```python
turtle.xcor()
```

turtle. `ycor()`
Return the turtle's y coordinate.

```python
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28, 76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

```python
turtle.ycor()
```

turtle. `heading()`
Return the turtle's current heading (value depends on the turtle mode, see `mode()`).

```python
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00, 86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

```python
turtle.heading()
```

turtle. `distance(x, y=None)`
Parameters:

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if x is a number, else **None**

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```python
>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.**degrees**(*fullcircle*=360.0)

Parameters:  **fullcircle** – a number

Set angle measurement units, i.e. set number of “degrees” for a full circle. Default value is 360 degrees.

```python
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
```

Change angle measurement unit to grad (also known grade, or gradian and equals 1/100-th of the right angle).

```python
>>> turtle.degrees(400.0)
>>> turtle.heading()
```
turtle. **radians()**

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```python
>>> turtle.radians()
```

```
90.0
```

**Pen control**

**Drawing state**

turtle. **pendown()**
turtle. **pd()**
turtle. **down()**

Pull the pen down – drawing when moving.

turtle. **penup()**
turtle. **pu()**
turtle. **up()**

Pull the pen up – no drawing when moving.

turtle. **pensize(width=None)**
turtle. **width(width=None)**
**Parameters:** width – a positive number

Set the line thickness to `width` or return it. If resizemode is set to “auto” and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```python
>>> turtle.pensize()
1
>>> turtle.pensize(10)  # from here on lines of
```

**turtle.pen**(pen=None, **pendict)**

**Parameters:**

- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen’s attributes in a “pen-dictionary” with the following key/value pairs:

- “shown”: True/False
- “pendown”: True/False
- “pencolor”: color-string or color-tuple
- “fillcolor”: color-string or color-tuple
- “pensize”: positive number
- “speed”: number in range 0..10
- “resizemode”: “auto” or “user” or “noresize”
- “stretchfactor”: (positive number, positive number)
- “outline”: positive number
- “tilt”: number

This dictionary can be used as argument for a subsequent call to `pen()` to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can
be used to set several pen attributes in one statement.

```python
>>> turtle.pen(fillcolor="black", pencolor="red",
>>>       sorted(turtle.pen().items())
[['fillcolor', 'black'], ['outline', 1], ['pencolor', ('pendown', True), ('pensize', 10), ('resizemode', 'shearfactor', 0.0), ('shown', True), ('speed', 'stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", ")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[['fillcolor', ''], ['outline', 1], ['pencolor',
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[['fillcolor', 'green'], ['outline', 1], ['pencolor',
```
pencolor()

Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

pencolor(colorstring)

Set pencolor to colorstring, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

pencolor((r, g, b))

Set pencolor to the RGB color represented by the tuple of r, g, and b. Each of r, g, and b must be in the range 0..colormode, where colormode is either 1.0 or 255 (see colormode()).

pencolor(r, g, b)

Set pencolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```python
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
```
turtle. **fillcolor**(*args*)

Return or set the fillcolor.

Four input formats are allowed:

**fillcolor()**

Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

**fillcolor**(colorstring)

Set fillcolor to *colorstring*, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

**fillcolor**((r, g, b))

Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see **colormode**()).

**fillcolor**(r, g, b)

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```python
>>> turtle.fillcolor("violet")
'violet'
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143)) # Integers,
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
```
turtle.color(*args)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color()

Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by pencolor() and fillcolor().

color(colorstring), color((r, g, b)), color(r, g, b)

Inputs as in pencolor(), set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1, g1, b1), (r2, g2, b2))

Equivalent to pencolor(colorstring1) and fillcolor(colorstring2) and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.
See also: Screen method `colormode()`.

Filling

turtle.\texttt{filling()}

Return fillstate (\texttt{True} if filling, \texttt{False} else).

```
>>> turtle.begin_fill()
>>> \textbf{if} turtle.filling():
...   turtle.pensize(5)
... \textbf{else}:
...   turtle.pensize(3)
```

turtle.\texttt{begin\_fill()}

To be called just before drawing a shape to be filled.

turtle.\texttt{end\_fill()}

Fill the shape drawn after the last call to `begin_fill()`.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.\texttt{reset()}

Delete the turtle’s drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
```
turtle.h**e**ad**i**ng()  
100.0  
>>> turtle.reset()  
>>> turtle.position()  
(0.00, 0.00)  
>>> turtle.h**e**ad**i**ng()  
0.0

turtle.c**l**ear()

Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.w**r**i**t**e(arg, move=False, align="left", font= ("Arial", 8, "normal"))

**Parameters:**
- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings “left”, “center” or right"
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of **arg** - at the current turtle position according to **align** (“left”, “center” or right”) and with the given font. If **move** is true, the pen is moved to the bottom-right corner of the text. By default, **move** is **False**.

>>> turtle.write("Home = ", True, align="center")  
>>> turtle.write((0,0), True)

**Turtle state**

Visibility
turtle.**hideturtle**()
turtle.**ht**()

Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```python
>>> turtle.hideturtle()
```

turtle.**showturtle**()
turtle.**st**()

Make the turtle visible.

```python
>>> turtle.showturtle()
```

turtle.**isvisible**()

Return **True** if the Turtle is shown, **False** if it's hidden.

```python
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

**Appearance**

turtle.**shape**(name=None)

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”,
“classic”. To learn about how to deal with shapes see Screen method `register_shape()`.

```python
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(rmode=\texttt{None})

| Parameters: | rmode – one of the strings “auto”, “user”, “noresize” |

Set resizemode to one of the values: “auto”, “user”, “noresize”. If \texttt{rmode} is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapts the appearance of the turtle corresponding to the value of pensize.
- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by `shapesize()`.
- “noresize”: no adaption of the turtle’s appearance takes place.

resizemode(“user”) is called by `shapesize()` when used with arguments.

```python
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(stretch_wid=\texttt{None}, stretch_len=\texttt{None}, outline=\texttt{None})
turtle.**turtlesize**(*stretch_wid=None, stretch_len=None, outline=None*)

<table>
<thead>
<tr>
<th>Parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>stretch_wid</td>
<td>– positive number</td>
</tr>
<tr>
<td>stretch_len</td>
<td>– positive number</td>
</tr>
<tr>
<td>outline</td>
<td>– positive number</td>
</tr>
</tbody>
</table>

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: `stretch_wid` is stretchfactor perpendicular to its orientation, `stretch_len` is stretchfactor in direction of its orientation, `outline` determines the width of the shapes’s outline.

```python
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

| Parameters: | shear – number (optional) |

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i.e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.
```python
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(angle)

**Parameters:**  **angle** – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(angle)

**Parameters:**  **angle** – a number

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle's heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```
Deprecated since version 3.1.

```python
turtle.tiltangle(angle=None)
```

**Parameters:**
- **angle** – a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do not change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i.e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

```python
turtle.shapetransform(t11=None, t12=None, t21=None, t22=None)
```

**Parameters:**
- **t11** – a number (optional)
- **t12** – a number (optional)
- **t21** – a number (optional)
- **t22** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, 22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify
stretchfactor, shearfactor and tiltangle according to the given matrix.

```python
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform((4.0, -1.0, -0.0, 2.0)
```

turtle. **get_shapepoly()**

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```python
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

**Using events**

turtle. **onclick**(fun, btn=1, add=None)

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fun</strong> – a function with two arguments which will be called with the coordinates of the clicked point on the canvas</td>
</tr>
<tr>
<td><strong>btn</strong> – number of the mouse-button, defaults to 1 (left mouse button)</td>
</tr>
<tr>
<td><strong>add</strong> – True or False – if True, a new binding will be added, otherwise it will replace a former binding</td>
</tr>
</tbody>
</table>

Bind *fun* to mouse-click events on this turtle. If *fun* is None, existing
bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```python
>>> def turn(x, y):
...     left(180)
... 
>>> onclick(turn)  # Now clicking into the turtle
>>> onclick(None)  # event-binding will be remove
```

`turtle. onrelease(fun, btn=1, add=None)``

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas</td>
</tr>
<tr>
<td>btn – number of the mouse-button, defaults to 1 (left mouse button)</td>
</tr>
<tr>
<td>add – True or False – if True, a new binding will be added, otherwise it will replace a former binding</td>
</tr>
</tbody>
</table>

Bind `fun` to mouse-button-release events on this turtle. If `fun` is `None`, existing bindings are removed.

```python
>>> class MyTurtle(Turtle):
...     def glow(self, x, y):
...         self.fillcolor("red")
...     def unglow(self, x, y):
...         self.fillcolor("")
... 
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)  # clicking on
>>> turtle.onrelease(turtle.unglow)  # releasing t
```

`turtle. ondrag(fun, btn=1, add=None)```
Parameters:

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is *None*, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```python
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

**Special Turtle methods**

- **turtle.begin_poly()**
  Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

- **turtle.end_poly()**
  Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

- **turtle.get_poly()**
  Return the last recorded polygon.
```python
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone()**
Create and return a clone of the turtle with same position, heading and turtle properties.

```python
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle()**  
turtle.**getpen()**
Return the Turtle object itself. Only reasonable use: as a function to return the “anonymous turtle”:

```python
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen()**
Return the **TurtleScreen** object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```python
>>> ts = turtle.getscreen()
>>> ts
```
turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")

turtle.setundobuffer(size)

**Parameters:**

| size – an integer or None |

Set or disable undobuffer. If *size* is an integer an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the `undo()` method/function. If *size* is `None`, the undobuffer is disabled.

```python
>>> turtle.setundobuffer(42)
```

turtle.undobufferentries()

Return number of entries in the undobuffer.

```python
>>> while undobufferentries():
...    undo()
```

### Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class `Shape` explicitly as described below:

1. Create an empty `Shape` object of type “compound”.

2. Add as many components to this object as desired, using the `addcomponent()` method.

For example:

```python
>>> s = Shape("compound")
```
3. Now add the Shape to the Screen’s shapelist and use it:

```python
>>> register_shape("myshape", s)
>>> shape("myshape")
```

**Note:** The **Shape** class is used internally by the **register_shape()** method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!
Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

turtle. bgcolor(*args)

| Parameters: | args – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers |

Set or return background color of the TurtleScreen.

```python
>>> screen.bgcolor("orange")
'orange'
>>> screen.bgcolor()
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()  
(128.0, 0.0, 128.0)
```

turtle. bgpic(picname=None)

| Parameters: | picname – a string, name of a gif-file or "nopic", or None |

Set background image or return name of current background image. If `picname` is a filename, set the corresponding image as background. If `picname` is "nopic", delete background image, if present. If `picname` is `None`, return the filename of the current background image.
turtle\texttt{.clear()}
turtle\texttt{.clearscreen()}

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

\textbf{Note:} This TurtleScreen method is available as a global function only under the name \texttt{clearscreen}. The global function \texttt{clear} is a different one derived from the Turtle method \texttt{clear}.

turtle\texttt{.reset()}
turtle\texttt{.resetscreen()}

Reset all Turtles on the Screen to their initial state.

\textbf{Note:} This TurtleScreen method is available as a global function only under the name \texttt{resetscreen}. The global function \texttt{reset} is another one derived from the Turtle method \texttt{reset}.

turtle\texttt{.screensize(canvwidth=\texttt{None}, canvheight=\texttt{None}, bg=\texttt{None})}

\begin{itemize}
  \item \textbf{canvwidth} – positive integer, new width of canvas in pixels
  \item \textbf{canvheight} – positive integer, new height of canvas in pixels
  \item \textbf{bg} – colorstring or color-tuple, new background color
\end{itemize}
If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```python
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000, 1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)  

turtle.**setworldcoordinates**(llx, lly, urx, ury)

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● <strong>llx</strong> – a number, x-coordinate of lower left corner of canvas</td>
</tr>
<tr>
<td>● <strong>lly</strong> – a number, y-coordinate of lower left corner of canvas</td>
</tr>
<tr>
<td>● <strong>urx</strong> – a number, x-coordinate of upper right corner of canvas</td>
</tr>
<tr>
<td>● <strong>ury</strong> – a number, y-coordinate of upper right corner of canvas</td>
</tr>
</tbody>
</table>

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

**ATTENTION:** in user-defined coordinate systems angles may appear distorted.

```python
>>> screen.reset()
```
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...    left(10)
...
>>> for _ in range(8):
...    left(45); fd(2)  # a regular octagon

Animation control

turtle.\texttt{delay}(\texttt{delay=\texttt{None}})

\begin{verbatim}
| Parameters:       | delay – positive integer |
\end{verbatim}

Set or return the drawing \texttt{delay} in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

\begin{verbatim}
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
\end{verbatim}

turtle.\texttt{tracer}(n=\texttt{None}, delay=\texttt{None})

\begin{verbatim}
| Parameters:       |       |
|                   | n – nonnegative integer |
|                   | delay – nonnegative integer |
\end{verbatim}

Turn turtle animation on/off and set delay for update drawings. If \texttt{n} is given, only each \texttt{n}-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of \texttt{n}. Second argument sets delay value (see \texttt{delay()}).
turtle.update()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method speed().

Using screen events

turtle.listen(xdummy=None, ydummy=None)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass listen() to the onclick method.

turtle.onkey(fun, key)
turtle.onkeyrelease(fun, key)

Parameters:
- fun – a function with no arguments or None
- key – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind fun to key-release event of key. If fun is None, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method listen().)
turtle.\texttt{onkeypress}(\texttt{fun}, \texttt{key}=\texttt{None})

<table>
<thead>
<tr>
<th>Parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fun –</td>
<td>a function with no arguments or None</td>
</tr>
<tr>
<td>key –</td>
<td>a string: key (e.g. “a”) or key-symbol (e.g. “space”)</td>
</tr>
</tbody>
</table>

Bind \texttt{fun} to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method \texttt{listen}().)

\begin{verbatim}
>>> def f():
...   fd(50)
... >>> screen.onkey(f, "Up")
>>> screen.listen()
\end{verbatim}

turtle.\texttt{onclick}(\texttt{fun}, \texttt{btn}=1, \texttt{add}=\texttt{None})
turtle.\texttt{onscreenclick}(\texttt{fun}, \texttt{btn}=1, \texttt{add}=\texttt{None})

<table>
<thead>
<tr>
<th>Parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fun –</td>
<td>a function with two arguments which will be called with the coordinates of the clicked point on the canvas</td>
</tr>
<tr>
<td>btn –</td>
<td>number of the mouse-button, defaults to 1 (left mouse button)</td>
</tr>
<tr>
<td>add –</td>
<td>\texttt{True} or \texttt{False} – if \texttt{True}, a new binding will be added, otherwise it will replace a former binding</td>
</tr>
</tbody>
</table>

Bind \texttt{fun} to mouse-click events on this screen. If \texttt{fun} is \texttt{None}, existing bindings are removed.
Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

>>> screen.onclick(turtle.goto)  # Subsequently click into the TurtleScreen
>>>                      # make the turtle move to the clicked point.
>>> screen.onclick(None)       # remove event binding again

**Note:** This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

`turtle.ontimer(fun, t=0)`

**Parameters:**
- `fun` – a function with no arguments
- `t` – a number >= 0

Install a timer that calls `fun` after `t` milliseconds.

```python
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...     screen.ontimer(f, 250)
>>> f()  ### makes the turtle march around
>>> running = False
```

`turtle.mainloop()`

`turtle.done()`

Starts event loop - calling Tkinter’s `mainloop` function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in `-n` mode (No subprocess) - for
interactive use of turtle graphics.

```python
>>> screen.mainloop()
```

**Input methods**

turtle.**textinput**(``title``, ``prompt``)

**Parameters:**
- ``title`` – string
- ``prompt`` – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```python
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(``title``, ``prompt``, ``default``=``None``, ``minval``=``None``, ``maxval``=``None``)

**Parameters:**
- ``title`` – string
- ``prompt`` – string
- ``default`` – number (optional)
- ``minval`` – number (optional)
- ``maxval`` – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the
number input. If the dialog is canceled, return None.

```python
>>> screen.numinput("Poker", "Your stakes:", 1000
```

**Settings and special methods**

turtle.\texttt{mode}(\texttt{mode}=	exttt{None})

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>\texttt{mode} – one of the strings “standard”, “logo” or “world”</th>
</tr>
</thead>
</table>

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old \texttt{turtle}. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. \textbf{Attention:} in this mode angles appear distorted if \texttt{x/y} unit-ratio doesn’t equal 1.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial turtle heading</th>
<th>positive angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>“standard”</td>
<td>to the right (east)</td>
<td>counterclockwise</td>
</tr>
<tr>
<td>“logo”</td>
<td>upward (north)</td>
<td>clockwise</td>
</tr>
</tbody>
</table>

```python
>>> mode("logo")  # resets turtle heading to nor
>>> mode()
'logo'
```

turtle.\texttt{colormode}(\texttt{cmode}=	exttt{None})

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>\texttt{cmode} – one of the values 1.0 or 255</th>
</tr>
</thead>
</table>

Return the colormode or set it to 1.0 or 255. Subsequently \texttt{r}, \texttt{g}, \texttt{b}
values of color triples have to be in the range $0..cmode$.

```python
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
  ...  
TurtleGraphicsError: bad color sequence: (240, 16
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.``getcanvas``()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```python
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.``getshapes``()

Return a list of names of all currently available turtle shapes.

```python
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.``register_shape``(name, shape=None)
turtle.``addshape``(name, shape=None)

There are three different ways to call this function:

1. name is the name of a gif-file and shape is None: Install the corresponding image shape.
>>> screen.register_shape("turtle.gif")

**Note:** Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

2. *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```python
>>> screen.register_shape("triangle", ((5, -3),
  ..., (5, 3),
  ..., (0, 3)))
```

3. *name* is an arbitrary string and *shape* is a (compound) `Shape` object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`. 

`turtle.turtles()`
Return the list of turtles on the screen.

```python
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.window_height()`
Return the height of the turtle window.

```python
>>> screen.window_height()
480
```

`turtle.window_width()`
Return the width of the turtle window.

```python
>>> screen.window_width()
```
Methods specific to Screen, not inherited from TurtleScreen

turtle.**bye**()
Shut the turtlegraphics window.

turtle.**exitonclick**()
Bind bye() method to mouse clicks on the Screen.

If the value “using_IDLE” in the configuration dictionary is False (default value), also enter mainloop. Remark: If IDLE with the -n switch (no subprocess) is used, this value should be set to True in turtle.cfg. In this case IDLE’s own mainloop is active also for the client script.

turtle.**setup**(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])
Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a turtle.cfg file.

**Parameters:**

- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if None, center window
- **startx** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window horizontally

```
>>> screen.setup (width=200, height=200, startx=0)
>>> # sets window to 200x200 pixels,
>>> screen.setup(width=.75, height=0.5, startx=None)
>>> # sets window to 75% of screen b
```

**turtle.title(titlestring)**

**Parameters:**

- **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```
Public classes

```python
class turtle.RawTurtle(canvas)
class turtle.RawPen(canvas)
```

| Parameters: | canvas – a tkinter.Canvas, a ScrolledCanvas or a TurtleScreen |

Create a turtle. The turtle has all methods described above as “methods of Turtle/RawTurtle”.

```python
class turtle.Turtle
```

Subclass of RawTurtle, has the same interface but draws on a default Screen object created automatically when needed for the first time.

```python
class turtle.TurtleScreen(cv)
```

| Parameters: | cv – a tkinter.Canvas |

Provides screen oriented methods like setbg() etc. that are described above.

```python
class turtle.Screen
```

Subclass of TurtleScreen, with four methods added.

```python
class turtle.ScrolledCanvas(master)
```

| Parameters: | master – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added |

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.
```python
class turtle.Shape(type_, data)

Parameters:
   type_ – one of the strings “polygon”, “image”, “compound”

Data structure modeling shapes. The pair (type_, data) must follow this specification:

<table>
<thead>
<tr>
<th>type_</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>“polygon”</td>
<td>a polygon-tuple, i.e. a tuple of pairs of coordinates</td>
</tr>
<tr>
<td>“image”</td>
<td>an image (in this form only used internally!)</td>
</tr>
<tr>
<td>“compound”</td>
<td>None (a compound shape has to be constructed using the addcomponent() method)</td>
</tr>
</tbody>
</table>

addcomponent(poly, fill, outline=None)

Parameters:
   poly – a polygon, i.e. a tuple of pairs of numbers
   fill – a color the poly will be filled with
   outline – a color for the poly’s outline (if given)

Example:

```python
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See Compound shapes.

```python
class turtle(Vec2D(x, y))

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics
programs too. Derived from tuple, so a vector is a tuple!

Provides (for $a, b$ vectors, $k$ number):

- $a + b$ vector addition
- $a - b$ vector subtraction
- $a * b$ inner product
- $k * a$ and $a * k$ multiplication with scalar
- $\text{abs}(a)$ absolute value of $a$
- $a.\text{rotate}(\text{angle})$ rotation
Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.

- Calling `help()` on methods or functions displays the docstrings:

```python
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three in the range 0..colormode or a 3-tuple of such

>>> screen.bgcolor("orange")
>>> screen.bgcolor()  
"orange"
>>> screen.bgcolor(0.5,0,0.5)
>>> screen.bgcolor()
"#800080"
```

```python
>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.
```
Aliases: penup | pu | up

No argument

>>> turtle.penup()

- The docstrings of the functions which are derived from methods have a modified form:

```python
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such
    numbers.

Example::

    >>> bgcolor("orange")
    >>> bgcolor()
    "orange"
    >>> bgcolor(0.5,0,0.5)
    >>> bgcolor()
    "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up
```
No argument

Example:
>>> penup()

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

```
turtle.write_docstringdict(filename="turtle_docstringdict"

Parameters: filename – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script `filename.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use turtle with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your turtle.cfg file this dictionary will be read in at import time and will replace the original English docstrings.
At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```plaintext
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = nosize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
```
The first four lines correspond to the arguments of the `Screen.setup()` method.

Line 5 and 6 correspond to the arguments of the method `Screen.screensize()`.

`shape` can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.

If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg-file).

If you want to reflect the turtle its state, you have to use `resizemode = auto`.

If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`.

The entries `exampleturtle` and `examplescreen` define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.

`using_IDLE`: Set this to `True` if you regularly work with IDLE and its `-n` switch (“no subprocess”). This will prevent `exitonclick()` to enter the mainloop.

There can be a `turtle.cfg` file in the directory where `turtle` is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the
demos (preferably not from within the demo-viewer).
**turtledemo — Demo scripts**

The *turtledemo* package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The *turtledemo* package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A *turtle.cfg* file which serves as an example of how to write and use such files.

The demo scripts are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytedesign</td>
<td>complex classical turtle graphics pattern</td>
<td>tracer(), delay, update()</td>
</tr>
<tr>
<td>chaos</td>
<td>graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the world coordinates</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>common sense expectations</th>
<th>clock</th>
<th>analog clock showing time of your computer</th>
<th>turtles as clock’s hands, on timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>colormixer</td>
<td>experiment with r, g, b</td>
<td>ondrag()</td>
<td></td>
</tr>
<tr>
<td>forest</td>
<td>3 breadth-first trees</td>
<td>randomization</td>
<td></td>
</tr>
<tr>
<td>fractalcurves</td>
<td>Hilbert &amp; Koch curves</td>
<td>recursion</td>
<td></td>
</tr>
<tr>
<td>lindenmayer</td>
<td>ethnomathematics (indian kolams)</td>
<td>L-System</td>
<td></td>
</tr>
<tr>
<td>minimal_hanoi</td>
<td>Towers of Hanoi</td>
<td>Rectangular Turtles as Hanoi discs (shape, shapesize)</td>
<td></td>
</tr>
<tr>
<td>nim</td>
<td>play the classical nim game with three heaps of sticks against the computer.</td>
<td>turtles as nimsticks, event driven (mouse, keyboard)</td>
<td></td>
</tr>
<tr>
<td>paint</td>
<td>super minimalistic drawing program</td>
<td>onclick()</td>
<td></td>
</tr>
<tr>
<td>peace</td>
<td>elementary</td>
<td>turtle: appearance and animation</td>
<td></td>
</tr>
<tr>
<td>penrose</td>
<td>aperiodic tiling with kites and darts</td>
<td>stamp()</td>
<td></td>
</tr>
<tr>
<td>planet_and_moon</td>
<td>simulation of gravitational system</td>
<td>compound shapes, Vec2D</td>
<td></td>
</tr>
<tr>
<td>round_dance</td>
<td>dancing turtles rotating pairwise in opposite direction</td>
<td>compound shapes, clone shapesize, tilt, get_shapepoly, update</td>
<td></td>
</tr>
<tr>
<td>sorting_animate</td>
<td>visual demonstration of different sorting methods</td>
<td>simple alignment, randomization</td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td>a (graphical) breadth first tree (using generators)</td>
<td>clone()</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>two_canvases</td>
<td>simple design</td>
<td>turtles on two canvases</td>
<td></td>
</tr>
<tr>
<td>wikipedia</td>
<td>a pattern from the wikipedia article on turtle graphics</td>
<td>clone(), undo()</td>
<td></td>
</tr>
<tr>
<td>yinyang</td>
<td>another elementary example</td>
<td>circle()</td>
<td></td>
</tr>
</tbody>
</table>

Have fun!
Changes since Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen/Screen`-methods.)

- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.

- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.
Changes since Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with Screen and Turtle objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.
**cmd** — Support for line-oriented command interpreters

Source code: Lib/cmd.py

The **Cmd** class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

```python
class cmd.Cmd(completekey='tab', stdin=None, stdout=None)
```

A **Cmd** instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate **Cmd** itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit **Cmd**'s methods and encapsulate action methods.

The optional argument `completekey` is the `readline` name of a completion key; it defaults to `tab`. If `completekey` is not `None` and `readline` is available, command completion is done automatically.

The optional arguments `stdin` and `stdout` specify the input and output file objects that the **Cmd** instance or subclass instance will use for input and output. If not specified, they will default to `sys.stdin` and `sys.stdout`.

If you want a given `stdin` to be used, make sure to set the instance's `use_rawinput` attribute to `False`, otherwise `stdin` will
be ignored.
Cmd Objects

A **Cmd** instance has the following methods:

**Cmd.cmdloop(intro=None)**

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the `intro` class attribute).

If the **readline** module is loaded, input will automatically inherit **bash**-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character `?` is dispatched to the method `do_help()`. As another special case, a line beginning with the character `!` is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the **postcmd()** method returns a true value. The `stop` argument to **postcmd()** is the return value from the command's corresponding `do_*()` method.

If completion is enabled, completing commands will be done...
automatically, and completing of commands args is done by calling complete_foo() with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match: all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of **Cmd** inherit a predefined do_help(). This method, called with an argument `'bar'`, invokes the corresponding method help_bar(), and if that is not present, prints the docstring of do_bar(), if available. With no argument, do_help() lists all available help topics (that is, all commands with corresponding help_*( ) methods or commands that have docstrings), and also lists any undocumented commands.

**Cmd**. onecmd(str)
Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the precmd() and postcmd() methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a do_*( ) method for the command *str*, the return value of that method is returned, otherwise the return value from the default() method is returned.

**Cmd**. emptyline()
Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.
**Cmd. default**(line)

Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

**Cmd. completedefault**(text, line, begidx, endidx)

Method called to complete an input line when no command-specific complete_*() method is available. By default, it returns an empty list.

**Cmd. precmd**(line)

Hook method executed just before the command line line is interpreted, but after the input prompt is generated and issued. This method is a stub in Cmd; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the onecmd() method; the precmd() implementation may re-write the command or simply return line unchanged.

**Cmd. postcmd**(stop, line)

Hook method executed just after a command dispatch is finished. This method is a stub in Cmd; it exists to be overridden by subclasses. line is the command line which was executed, and stop is a flag which indicates whether execution will be terminated after the call to postcmd(); this will be the return value of the onecmd() method. The return value of this method will be used as the new value for the internal flag which corresponds to stop; returning false will cause interpretation to continue.

**Cmd. preloop()**

Hook method executed once when cmdloop() is called. This method is a stub in Cmd; it exists to be overridden by subclasses.
Cmd. **postloop()**

Hook method executed once when `cmdloop()` is about to return. This method is a stub in Cmd; it exists to be overridden by subclasses.

Instances of **Cmd** subclasses have some public instance variables:

**Cmd. prompt**

The prompt issued to solicit input.

**Cmd. identchars**

The string of characters accepted for the command prefix.

**Cmd. lastcmd**

The last nonempty command prefix seen.

**Cmd. cmdqueue**

A list of queued input lines. The cmdqueue list is checked in `cmdloop()` when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

**Cmd. intro**

A string to issue as an intro or banner. May be overridden by giving the `cmdloop()` method an argument.

**Cmd. doc_header**

The header to issue if the help output has a section for documented commands.

**Cmd. misc_header**

The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*()` methods without corresponding `do_*( )` methods).
**Cmd. undoc_header**

The header to issue if the help output has a section for undocumented commands (that is, there are do_*() methods without corresponding help_*() methods).

**Cmd. ruler**

The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to '='.

**Cmd. use_rawinput**

A flag, defaulting to true. If true, cmdloop() uses input() to display a prompt and read the next command; if false, sys.stdout.write() and sys.stdin.readline() are used. (This means that by importing readline, on systems that support it, the interpreter will automatically support Emacs-like line editing and command-history keystrokes.)
Cmd Example

The `cmd` module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the `turtle` module.

Basic turtle commands such as `forward()` are added to a `Cmd` subclass with method named `do_forward()`. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the `precmd()` method which is responsible for converting the input to lowercase and writing the commands to a file. The `do_playback()` method reads the file and adds the recorded commands to the `cmdqueue` for immediate playback:

```python
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell. Type help or ? to list commands.'
    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:'
        pass
```
right(*parse(arg))

def do_left(self, arg):
    'Turn turtle left by given number of degrees'
    left(*parse(arg))

def do_goto(self, arg):
    'Move turtle to an absolute position with changing orientation.'
    goto(*parse(arg))

def do_home(self, arg):
    'Return turtle to the home position: HOME'
    home()

def do_circle(self, arg):
    'Draw circle with given radius and options except filling.
    circle(*parse(arg))'

def do_position(self, arg):
    'Print the current turtle position: POSITION'
    print('Current position is %d %d\n' % position)

def do_heading(self, arg):
    'Print the current turtle heading in degrees'
    print('Current heading is %d\n' % (heading()))

def do_color(self, arg):
    'Set the color: COLOR BLUE'
    color(arg.lower())

def do_undo(self, arg):
    'Undo (repeatedly) the last turtle action(s)'*parse(arg))

def do_reset(self, arg):
    'Clear the screen and return turtle to center'
    reset()

def do_bye(self, arg):
    'Stop recording, close the turtle window, and exit.'
    print('Thank you for using Turtle')
    self.close()
    bye()
    return True

# ------ record and playback ------
def do_record(self, arg):
    'Save future commands to filename: RECORD
    return True

    def do_record(self, arg):
        'Save future commands to filename: RECORD
        return True
self.file = open(arg, 'w')

def do_playback(self, arg):
    'Playback commands from a file: PLAYBACK rose.cmd'
    self.close()
    with open(arg) as f:
        self.cmdqueue.extend(f.read().splitlines)

def precmd(self, line):
    line = line.lower()
    if self.file and 'playback' not in line:
        print(line, file=self.file)
    return line

def close(self):
    if self.file:
        self.file.close()
        self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

Welcome to the turtle shell. Type help or ? to list commands.
(turtle) ?

Documented commands (type help <topic>):
-----------------------------------------------
bye    color    goto    home    playback    record    right
circle forward heading left    position    reset    undo
(turtle) help forward
Move the turtle forward by the specified distance:
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
shlex — Simple lexical analysis

Source code: Lib/shlex.py

The shlex class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The shlex module defines the following functions:

shlex.split(s, comments=False, posix=True)
Split the string s using shell-like syntax. If comments is False (the default), the parsing of comments in the given string will be disabled (setting the commenters attribute of the shlex instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the posix argument is false.

**Note:** Since the split() function instantiates a shlex instance, passing None for s will read the string to split from standard input.

shlex.quote(s)
Return a shell-escaped version of the string s. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```python
>>> filename = 'somefile; rm -rf ~'
```
`quote()` lets you plug the security hole:

```python
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l ''''somefile; rm -rf ~'''''
```

The quoting is compatible with UNIX shells and with `split()`:

```python
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', 'ls -l ''''somefile; rm -rf ~'''''
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

New in version 3.3.

The `shlex` module defines the following class:

```python
class shlex.shlex(instream=None, infile=None, posix=False, punctuation_chars=False)
```

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given,
input will be taken from \texttt{sys.stdin}. The second optional argument is a filename string, which sets the initial value of the \texttt{inlines} attribute. If the \texttt{instream} argument is omitted or equal to \texttt{sys.stdin}, this second argument defaults to “stdin”. The \texttt{posix} argument defines the operational mode: when \texttt{posix} is not true (default), the \texttt{shlex} instance will operate in compatibility mode. When operating in POSIX mode, \texttt{shlex} will try to be as close as possible to the POSIX shell parsing rules. The \texttt{punctuation_chars} argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, \texttt{False}, preserves the behaviour seen under Python 3.5 and earlier. If set to \texttt{True}, then parsing of the characters \texttt{\{()<>|&} is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the \texttt{wordchars} attribute that appear in \texttt{punctuation_chars} will be removed from \texttt{wordchars}. See \texttt{Improved Compatibility with Shells} for more information.

\textit{Changed in version 3.6:} The \texttt{punctuation_chars} parameter was added.

\textbf{See also:}

\textbf{Module \texttt{configparser}}

Parser for configuration files similar to the Windows \texttt{.ini} files.
shlex Objects

A `shlex` instance has the following methods:

**shlex.get_token()**
Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `eof` is returned (the empty string ('') in non-POSIX mode, and `None` in POSIX mode).

**shlex.push_token(str)**
Push the argument onto the token stack.

**shlex.read_token()**
Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

**shlex.sourcehook(filename)**
When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles
#include "file.h".

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a shlex instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

```
shlex.push_source(newstream, newfile=None)
```

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook()` method.

```
shlex.pop_source()
```

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

```
shlex.error_leader(infile=None, lineno=None)
```

This method generates an error message leader in the format of a Unix C compiler error label; the format is ""%s", line %d: "", where the %s is replaced with the name of the current source file and the %d with the current input line number (the optional
arguments can be used to override these).

This convenience is provided to encourage shlex users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of shlex subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.commenters
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just '#' by default.

shlex.wordchars
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If punctuation_chars is not empty, the characters ~-./*?=, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in punctuation_chars will be removed from wordchars if they are present there.

shlex.whitespace
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.escape
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just '\\' by default.
shlex. **quotes**
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex. **escapedquotes**
Characters in **quotes** that will interpret escape characters defined in escape. This is only used in POSIX mode, and includes just `'"'` by default.

shlex. **whitespace_split**
If True, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with shlex, getting tokens in a similar way to shell arguments. If this attribute is True, punctuation_chars will have no effect, and splitting will happen only on whitespaces. When using punctuation_chars, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave whitespace_split as False (the default value).

shlex. **infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex. **instream**
The input stream from which this shlex instance is reading characters.

shlex. **source**
This attribute is None by default. If you assign a string to it, that
string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the `close()` method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

**shlex.debug**

If this attribute is numeric and 1 or more, a `shlex` instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

**shlex.lineno**

Source line number (count of newlines seen so far plus one).

**shlex.token**

The token buffer. It may be useful to examine this when catching exceptions.

**shlex.eof**

Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to `None` in POSIX mode.

**shlex.punctuation_chars**

Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, ‘>>>’ could be returned as a token, even though it may not be recognised as such by shells.

*New in version 3.6.*
Parsing Rules

When operating in non-POSIX mode, shlex will try to obey to the following rules.

- Quote characters are not recognized within words (`Do\"Not\"Separate` is parsed as the single word `DoNotSeparate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do\"Separate` is parsed as `"Do" and Separate`);
- If `whitespace_split` is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, shlex will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, shlex will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do\"Not\"Separate` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `\`\`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of `escapedquotes` (e.g. `"\"\") preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of
escapedquotes (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in escape. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.

- EOF is signaled with a None value;
- Quoted empty strings ('') are allowed.
Improved Compatibility with Shells

New in version 3.6.

The `shlex` class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```python
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"  
>>> list(shlex.shlex(text))
['a', '&', 'b', ';', 'c', '&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';', '(', 'def', '"ghi"', ')']

>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the `punctuation_chars` parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For
example:

```python
>>> import shlex
>>> s = shlex.shlex("a & b || c", punctuation_chars=\n>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

**Note:** When `punctuation_chars` is specified, the `wordchars` attribute is augmented with the characters ~-./*?=, That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```python
>>> import shlex
>>> s = shlex.shlex("~/a && b-c - --color=auto || d \n... punctuation_chars=True)
>>> list(s)
['~/a', '&', '&', 'b-c', ' --color=auto', ' ||', 'd', '*
```

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for `shlex`.)
Graphical User Interfaces with Tk

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the `tkinter` package, and its extension, the `tkinter.tix` and the `tkinter.ttk` modules.

The `tkinter` package is a thin object-oriented layer on top of Tcl/Tk. To use `tkinter`, you don’t need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. `tkinter` is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module `_tkinter` provides a threadsafe mechanism which allows Python and Tcl to interact.

`tkinter`'s chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. `tkinter` is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the Other Graphical User Interface Packages section.

- `tkinter` — Python interface to Tcl/Tk
  - Tkinter Modules
  - Tkinter Life Preserver
    - How To Use This Section
    - A Simple Hello World Program
  - A (Very) Quick Look at Tcl/Tk
  - Mapping Basic Tk into Tkinter
  - How Tk and Tkinter are Related
Handy Reference
- Setting Options
- The Packer
- Packer Options
- Coupling Widget Variables
- The Window Manager
- Tk Option Data Types
- Bindings and Events
- The index Parameter
- Images
- File Handlers

**tkinter.ttk — Tk themed widgets**
- Using Ttk
- Ttk Widgets
- Widget
  - Standard Options
  - Scrollable Widget Options
  - Label Options
  - Compatibility Options
  - Widget States
  - ttk.Widget
- Combobox
  - Options
  - Virtual events
  - ttk.Combobox
- Spinbox
  - Options
  - Virtual events
  - ttk.Spinbox
- Notebook
  - Options
  - Tab Options
  - Tab Identifiers
  - Virtual Events
• ttk.Notebook
  ○ Progressbar
    ■ Options
    ■ ttk.Progressbar
  ○ Separator
    ■ Options
  ○ Sizegrip
    ■ Platform-specific notes
    ■ Bugs
  ○ Treeview
    ■ Options
    ■ Item Options
    ■ Tag Options
    ■ Column Identifiers
    ■ Virtual Events
    ■ ttk.Treeview
  ○ Ttk Styling
    ■ Layouts
• tkinter.tix — Extension widgets for Tk
  ○ Using Tix
  ○ Tix Widgets
    ■ Basic Widgets
    ■ File Selectors
    ■ Hierarchical ListBox
    ■ Tabular ListBox
    ■ Manager Widgets
    ■ Image Types
    ■ Miscellaneous Widgets
    ■ Form Geometry Manager
  ○ Tix Commands
• tkinter.scrolledtext — Scrolled Text Widget
• IDLE
  ○ Menus
    ■ File menu (Shell and Editor)
- Edit menu (Shell and Editor)
- Format menu (Editor window only)
- Run menu (Editor window only)
- Shell menu (Shell window only)
- Debug menu (Shell window only)
- Options menu (Shell and Editor)
- Window menu (Shell and Editor)
- Help menu (Shell and Editor)
- Context Menus
  - Editing and navigation
    - Editor windows
    - Key bindings
    - Automatic indentation
    - Completions
    - Calltips
    - Code Context
    - Python Shell window
    - Text colors
  - Startup and code execution
    - Command line usage
    - Startup failure
    - Running user code
    - User output in Shell
    - Developing tkinter applications
    - Running without a subprocess
  - Help and preferences
    - Help sources
    - Setting preferences
    - IDLE on macOS
    - Extensions
- Other Graphical User Interface Packages
tkinter — Python interface to Tcl/Tk

Source code: Lib/tkinter/__init__.py

The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

See also: Tkinter documentation:

Python Tkinter Resources
The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

TKDocs
Extensive tutorial plus friendlier widget pages for some of the widgets.

Tkinter reference: a GUI for Python
On-line reference material.

Tkinter docs from effbot
Online reference for tkinter supported by effbot.org.
Programming Python
   Book by Mark Lutz, has excellent coverage of Tkinter.

Modern Tkinter for Busy Python Developers
   Book by Mark Rozerman about building attractive and modern graphical user interfaces with Python and Tkinter.

Python and Tkinter Programming

Tcl/Tk documentation:

Tk commands
   Most commands are available as tkinter or tkinter.ttk classes. Change ‘8.6’ to match the version of your Tcl/Tk installation.

Tcl/Tk recent man pages
   Recent Tcl/Tk manuals on www.tcl.tk.

ActiveState Tcl Home Page
   The Tk/Tcl development is largely taking place at ActiveState.

Tcl and the Tk Toolkit
   Book by John Ousterhout, the inventor of Tcl.

Practical Programming in Tcl and Tk
   Brent Welch’s encyclopedic book.
Tkinter Modules

Most of the time, `tkinter` is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named `_tkinter`. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, `tkinter` includes a number of Python modules, `tkinter.constants` being one of the most important. Importing `tkinter` will automatically import `tkinter.constants`, so, usually, to use Tkinter all you need is a simple import statement:

```python
import tkinter
```

Or, more often:

```python
from tkinter import *
```

```python
class tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)
    The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

tkinter.Tcl(screenName=None, baseName=None, className='Tk', useTk=0)
    The `Tcl()` function is a factory function which creates an object much like that created by the `Tk` class, except that it does not
initialize the Tk subsystem. This is most often useful when driving
the Tcl interpreter in an environment where one doesn’t want to
create extraneous toplevel windows, or where one cannot (such as
Unix/Linux systems without an X server). An object created by the
Tcl() object can have a Toplevel window created (and the Tk
subsystem initialized) by calling its loadtk() method.

Other modules that provide Tk support include:

**tkinter.scrolledtext**
Text widget with a vertical scroll bar built in.

**tkinter.colorchooser**
Dialog to let the user choose a color.

**tkinter.commondialog**
Base class for the dialogs defined in the other modules listed here.

**tkinter.filedialog**
Common dialogs to allow the user to specify a file to open or save.

**tkinter.font**
Utilities to help work with fonts.

**tkinter.messagebox**
Access to standard Tk dialog boxes.

**tkinter.simpledialog**
Basic dialogs and convenience functions.

**tkinter.dnd**
Drag-and-drop support for tkinter. This is experimental and
should become deprecated when it is replaced with the Tk DND.

**turtle**
Turtle graphics in a Tk window.
Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tk was written by John Ousterhout while at Berkeley.
- Tkinter was written by Steen Lumholt and Guido van Rossum.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The HTML rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form “how do I do blah”, it is often best to find out how to do “blah” in straight Tk, and then convert this back into the corresponding `tkinter` call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can’t fulfill that role, so the best we can do is point you to the best documentation that exists.
Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the `manN` directory are most useful. The `man3` man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.
- `tkinter/__init__.py` is a last resort for most, but can be a good place to go when nothing else makes sense.

### A Simple Hello World Program

```python
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there['text'] = 'Hello World
(click me)
        self.hi_there['command'] = self.say_hi
        self.hi_there.pack(side='top')

        self.quit = tk.Button(self, text='QUIT', fg='red'
        command=self.master.destroy)
        self.quit.pack(side='bottom')

    def say_hi(self):
```
print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

Notes:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren’t meant to be instantiated independently.
- The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The Widget class is not meant to be instantiated, it is meant only for subclassing to make “real” widgets (in C++, this is called an ‘abstract class’).

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section Mapping Basic Tk into Tkinter for the tkinter equivalents of what’s below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its class, the options that help configure it, and the actions that make it do useful things.

To make a widget in Tk, the command is always of the form:

classCommand newPathname options
**classCommand**

denotes which kind of widget to make (a button, a label, a menu...)

**newPathname**
is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with *pathnames*, just like files in a file system. The top level widget, the *root*, is called `.` (period) and children are delimited by more periods. For example, `.myApp.controlPanel.okButton` might be the name of a widget.

**options**
configure the widget’s appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a ‘-‘, like Unix shell command flags, and values are put in quotes if they are more than one word.

For example:

```
button .fred -fg red -text "hi there"
```

Once created, the pathname to the widget becomes a new command. This new *widget command* is the programmer’s handle for getting the new widget to perform some *action*. In C, you’d express this as `someAction(fred, someOptions)`, in C++, you would express this as `fred.someAction(someOptions)`, and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, `.fred`, starts with a dot.
As you’d expect, the legal values for *someAction* will depend on the widget’s class: `.fred disable` works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of *someOptions* is action dependent. Some actions, like `disable`, require no arguments, others, like a text-entry box’s `delete` command, would need arguments to specify what range of text to delete.
Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred           ======>  fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```
button .panel.fred     ======>  fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphenated tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section Setting Options on setting options.

```
button .fred -fg red   ======>  fred = Button(panel)
.fred configure -fg red ======>  fred["fg"] = red
 OR ==>  fred.config(fg=')
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in `tkinter/__init__.py`.

```
.fred invoke           ======>  fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this
functionality, and the various forms of the pack command are implemented as methods. All widgets in `tkinter` are subclassed from the Packer, and so inherit all the packing methods. See the `tkinter.tix` module documentation for additional information on the Form geometry manager.

```
pack .fred -side left =====> fred.pack(side='left')
```
How Tk and Tkinter are Related

From the top down:

Your App Here (Python)
   A Python application makes a tkinter call.

tkinter (Python Package)
   This call (say, for example, creating a button widget), is implemented in the tkinter package, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

ltkinter (C)
   These commands and their arguments will be passed to a C function in the _tkinter - note the underscore - extension module.

Tk Widgets (C and Tcl)
   This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python tkinter package is imported. (The user never sees this stage).

Tk (C)
   The Tk part of the Tk Widgets implement the final mapping to ...

Xlib (C)
   the Xlib library to draw graphics on the screen.
Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```python
fred = Button(self, fg="red", bg="blue")
```

After object creation, treating the option name like a dictionary index

```python
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```python
fred.config(fg="red", bg="blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list “STANDARD OPTIONS” and “WIDGET SPECIFIC OPTIONS” for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the `options(3)` man page.

No distinction between standard and widget-specific options is made in this document. Some options don’t apply to some kinds of widgets. Whether a given widget responds to a particular option depends on
the class of the widget; buttons have a command option, labels do not.

The options supported by a given widget are listed in that widget’s man page, or can be queried at runtime by calling the config() method without arguments, or by calling the keys() method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, 'relief') and whose values are 5-tuples.

Some options, like bg are synonyms for common options with long names (bg is shorthand for “background”). Passing the config() method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the “real” option (such as ('bg', 'background')).

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>option name</td>
<td>'relief'</td>
</tr>
<tr>
<td>1</td>
<td>option name for database lookup</td>
<td>'relief'</td>
</tr>
<tr>
<td>2</td>
<td>option class for database lookup</td>
<td>'Relief'</td>
</tr>
<tr>
<td>3</td>
<td>default value</td>
<td>'raised'</td>
</tr>
<tr>
<td>4</td>
<td>current value</td>
<td>'groove'</td>
</tr>
</tbody>
</table>

Example:

```python
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.
The Packer

The packer is one of Tk’s geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual master. In contrast to the more cumbersome placer (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - above, to the left of, filling, etc - and works everything out to determine the exact placement coordinates for you.

The size of any master widget is determined by the size of the “slave widgets” inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It’s a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer’s pack() method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```python
fred.pack()  # defaults to side =
fred.pack(side="left")
fred.pack(expand=1)
```
Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor
Anchor type. Denotes where the packer is to place each slave in its parcel.

expand
Boolean, 0 or 1.

fill
Legal values: 'x', 'y', 'both', 'none'.

ipadx and ipady
A distance - designating internal padding on each side of the slave widget.

padx and pady
A distance - designating external padding on each side of the slave widget.

side
Legal values are: 'left', 'right', 'top', 'bottom'.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.
Unfortunately, in the current implementation of `tkinter` it is not possible to hand over an arbitrary Python variable to a widget through a `variable` or `textvariable` option. The only kinds of variables for which this works are variables that are subclassed from a class called `Variable`, defined in `tkinter`.

There are many useful subclasses of `Variable` already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```python
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return
        # we will have the program print out the value of the application variable when the user hits return
        self.entrythingy.bind(<Key-Return>',
            self.print_contents)
```
def print_contents(self, event):
    print("hi. contents of entry is now --->",
          self.contents.get())

The Window Manager

In Tk, there is a utility command, \texttt{wm}, for interacting with the window manager. Options to the \texttt{wm} command allow you to control things like titles, placement, icon bitmaps, and the like. In \texttt{tkinter}, these commands have been implemented as methods on the Wm class. Toplevel widgets are subclassed from the Wm class, and so can call the Wm methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget’s master. Of course if the widget has been packed inside of a frame, the master won’t represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the \texttt{\_\_root()} method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```python
import tkinter as tk

class App(tk.Frame):
    def \_\_init\_\_(self, master=\texttt{None}):
        super().\_\_init\_\_(master)
        self.pack()

# create the application
myapp = App()
```
Tk Option Data Types

anchor
   Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "center".

bitmap
   There are eight built-in, named bitmaps: 'error', 'gray25', 'gray50', 'hourglass', 'info', 'questhead', 'question', 'warning'. To specify an X bitmap filename, give the full path to the file, preceded with an @, as in "@/usr/contrib/bitmap/gumby.bit".

boolean
   You can pass integers 0 or 1 or the strings "yes" or "no".

callback
   This is any Python function that takes no arguments. For example:

   ```python
   def print_it():
       print("hi there")
   fred["command"] = print_it
   ```

color
   Colors can be given as the names of X colors in the rgb.txt file, or

cursor
The standard X cursor names from cursorfont.h can be used, without the XC_ prefix. For example to get a hand cursor (XC_hand2), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance
Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: c for centimetres, i for inches, m for millimetres, p for printer's points. For example, 3.5 inches is expressed as "3.5i".

font
Tk uses a list font name format, such as {courier 10 bold}. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry
This is a string of the form widthxheight, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: fred["geometry"] = "200x100".

justify
Legal values are the strings: "left", "center", "right", and "fill".

region
This is a string with four space-delimited elements, each of which is
a legal distance (see above). For example: "2 3 4 5" and "3i 2i 4.5i 2i" and "3c 2c 4c 10.43c" are all legal regions.

relief
Determines what the border style of a widget will be. Legal values are: "raised", "sunken", "flat", "groove", and "ridge".

scrollcommand
This is almost always the set() method of some scrollbar widget, but can be any widget method that takes a single argument.

wrap:
Must be one of: "none", "char", or "word".

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```python
def bind(self, sequence, func, add=' '):
```

where:

sequence
is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

func
is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as callbacks.)

add
is optional, either '' or '+'. Passing an empty string denotes that this binding is to replace any other bindings that this event is
associated with. Passing a '+' means that this function is to be added to the list of functions bound to this event type.

For example:

```python
def turn_red(self, event):
    event.widget["activeforeground"] = "red"
self.button.bind("<Enter>", self.turn_red)
```

Notice how the widget field of the event is being accessed in the `turn_red()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

<table>
<thead>
<tr>
<th>Tk</th>
<th>Tkinter Event Field</th>
<th>Tk</th>
<th>Tkinter Event Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>focus</td>
<td>%A</td>
<td>char</td>
</tr>
<tr>
<td>%h</td>
<td>height</td>
<td>%E</td>
<td>send_event</td>
</tr>
<tr>
<td>%k</td>
<td>keycode</td>
<td>%K</td>
<td>keysym</td>
</tr>
<tr>
<td>%s</td>
<td>state</td>
<td>%N</td>
<td>keysym_num</td>
</tr>
<tr>
<td>%t</td>
<td>time</td>
<td>%T</td>
<td>type</td>
</tr>
<tr>
<td>%w</td>
<td>width</td>
<td>%W</td>
<td>widget</td>
</tr>
<tr>
<td>%x</td>
<td>x</td>
<td>%X</td>
<td>x_root</td>
</tr>
<tr>
<td>%y</td>
<td>y</td>
<td>%Y</td>
<td>y_root</td>
</tr>
</tbody>
</table>

The index Parameter

A number of widgets require “index” parameters to be passed. These are used to point at a specific place in a Text widget, or to particular
characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.)

Entry widgets have options that refer to character positions in the text being displayed. You can use these tkinter functions to access these special points in text widgets:

Text widget indexes

The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.)

Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string "active", which refers to the menu position that is currently under the cursor;
- the string "last" which refers to the last menu item;
- An integer preceded by @, as in @6, where the integer is interpreted as a y pixel coordinate in the menu’s coordinate system;
- the string "none", which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.
Images

Images of different formats can be created through the corresponding subclass of tkinter.Image:

- `BitmapImage` for images in XBM format.
- `PhotoImage` for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

**See also:** The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.
File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```python
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don’t know how many bytes are available for reading, you may not want to use the `BufferedIOBase` or `TextIOWrapper` read() or readline() methods, since these will insist on reading a predefined number of bytes. For sockets, the `recv()` or `recvfrom()` methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.createfilehandler(file, mask, func)`

Registers the file handler callback function `func`. The `file` argument may either be an object with a `fileno()` method (such as a file or socket object), or an integer file descriptor. The `mask` argument is an ORed combination of any of the three constants below. The callback is called as follows:

```python
callback(file, mask)
```
Widget.tk. **deletefilehandler**(file)

Unregisters a file handler.

**tkinter.** **READABLE**
**tkinter.** **WRITABLE**
**tkinter.** **EXCEPTION**

Constants used in the *mask* arguments.
tkinter.ttk — Tk themed widgets

Source code: Lib/tkinter/ttk.py

The tkinter.ttk module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if Tile has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for tkinter.ttk is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

See also:

Tk Widget Styling Support
A document introducing theming support for Tk
Using Ttk

To start using Ttk, import its module:

```python
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```python
from tkinter import *
from tkinter.ttk import *
```

That code causes several `tkinter.ttk` widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

**See also:**

**Converting existing applications to use Tile widgets**
A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.
Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar, and Spinbox. The other six are new: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. And all them are subclasses of Widget.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```python
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```python
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")
l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about TtkStyling, see the Style class documentation.
**Widget**

ttk.Widget defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

**Standard Options**

All the ttk Widgets accepts the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Specifies the window class. The class is used when querying the option database for the window’s other options, to determine the default bindtags for the window, and to select the widget’s default layout and style. This option is read-only, and may only be specified when the window is created.</td>
</tr>
<tr>
<td>cursor</td>
<td>Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.</td>
</tr>
<tr>
<td>takefocus</td>
<td>Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.</td>
</tr>
<tr>
<td>style</td>
<td>May be used to specify a custom widget style.</td>
</tr>
</tbody>
</table>

**Scrollable Widget Options**
The following options are supported by widgets that are controlled by a scrollbar.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscrollcommand</td>
<td>Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method Scrollbar.set() of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.</td>
</tr>
<tr>
<td>yscrollcommand</td>
<td>Used to communicate with vertical scrollbars. For some more information, see above.</td>
</tr>
</tbody>
</table>

Label Options

The following options are supported by labels, buttons and other button-like widgets.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Specifies a text string to be displayed inside the widget.</td>
</tr>
<tr>
<td>textvariable</td>
<td>Specifies a name whose value will be used in place of the text option resource. If set, specifies the index (0-based) of a character</td>
</tr>
<tr>
<td>underline</td>
<td>to underline in the text string. The underline character is used for mnemonic activation.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>image</td>
<td>Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by <code>Style.map()</code>, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.</td>
</tr>
</tbody>
</table>
| compound | Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are:  
  - text: display text only  
  - image: display image only  
  - top, bottom, left, right: display image above, below, left of, or right of the text, respectively.  
  - none: the default. display the image if present, otherwise the text. |
| width | If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used. |

### Compatibility Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>May be set to “normal” or “disabled” to control the “disabled” state bit. This is a write-only option: setting it changes the widget state, but the</td>
</tr>
</tbody>
</table>
**Widget States**

The widget state is a bitmap of independent state flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>The mouse cursor is over the widget and pressing a mouse button will cause some action to occur</td>
</tr>
<tr>
<td>disabled</td>
<td>Widget is disabled under program control</td>
</tr>
<tr>
<td>focus</td>
<td>Widget has keyboard focus</td>
</tr>
<tr>
<td>pressed</td>
<td>Widget is being pressed</td>
</tr>
<tr>
<td>selected</td>
<td>“On”, “true”, or “current” for things like Checkbuttons and radiobuttons</td>
</tr>
<tr>
<td>background</td>
<td>Windows and Mac have a notion of an “active” or foreground window. The background state is set for widgets in a background window, and cleared for those in the foreground window</td>
</tr>
<tr>
<td>readonly</td>
<td>Widget should not allow user modification</td>
</tr>
<tr>
<td>alternate</td>
<td>A widget-specific alternate display format</td>
</tr>
<tr>
<td>invalid</td>
<td>The widget’s value is invalid</td>
</tr>
</tbody>
</table>

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

**ttk.Widget**
Besides the methods described below, the ttk.Widget supports the methods tkinter.Widget.cget() and tkinter.Widget.configure().

class tkinter.ttk.Widget

identify(x, y)

Returns the name of the element at position x y, or the empty string if the point does not lie within any element.

x and y are pixel coordinates relative to the widget.

instate(statespec, callback=None, *args, **kw)

Test the widget's state. If a callback is not specified, returns True if the widget state matches statespec and False otherwise. If callback is specified then it is called with args if widget state matches statespec.

state(statespec=None)

Modify or inquire widget state. If statespec is specified, sets the widget state according to it and return a new statespec indicating which flags were changed. If statespec is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.
Combobox

The ttk.Combo box widget combines a text field with a pop-down list of values. This widget is a subclass of Entry.

Besides the methods inherited from Widget: Widget.cget(), Widget.configure(), Widget.identify(), Widget.instate() and Widget.state(), and the following inherited from Entry: Entry.bbox(), Entry.delete(), Entry.icursor(), Entry.index(), Entry.insert(), Entry.selection(), Entry.xview(), it has some other methods, described at ttk.Combo box.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exportselection</td>
<td>Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).</td>
</tr>
<tr>
<td>justify</td>
<td>Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.</td>
</tr>
<tr>
<td>height</td>
<td>Specifies the height of the pop-down listbox, in rows.</td>
</tr>
<tr>
<td>postcommand</td>
<td>A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.</td>
</tr>
<tr>
<td>state</td>
<td>One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>textvariable</td>
<td>Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See tkinter.StringVar.</td>
</tr>
<tr>
<td>values</td>
<td>Specifies the list of values to display in the drop-down listbox.</td>
</tr>
<tr>
<td>width</td>
<td>Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font.</td>
</tr>
</tbody>
</table>

Virtual events

The combobox widgets generates a <<ComboboxSelected>> virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.Combobox
current(newindex=None)

If newindex is specified, sets the combobox value to the element position newindex. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
Returns the current value of the combobox.

**set**(*value*)
Sets the value of the combobox to *value*. 
Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from `Widget`: `Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()` and `Widget.state()`, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as <code>from_</code> when used as an argument, since <code>from</code> is a Python keyword.</td>
</tr>
<tr>
<td>to</td>
<td>Float value. If set, this is the maximum value to which the increment button will increment.</td>
</tr>
<tr>
<td>increment</td>
<td>Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence of string or float values. If specified, the increment/decrement buttons will cycle</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>values</th>
<th>through the items in this sequence rather than incrementing or decrementing numbers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrap</td>
<td>Boolean value. If True, increment and decrement buttons will cycle from the to value to the from value or the from value to the to value, respectively.</td>
</tr>
<tr>
<td>format</td>
<td>String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form “%W.Pf”, where W is the padded width of the value, P is the precision, and ‘%’ and ‘f’ are literal.</td>
</tr>
<tr>
<td>command</td>
<td>Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.</td>
</tr>
</tbody>
</table>

**Virtual events**

The spinbox widget generates an **<<Increment>>** virtual event when the user presses <Up>, and a **<<Decrement>>** virtual event when the user presses <Down>.

**ttk.Spinbox**

```python
class tkinter.ttk.Spinbox
    get()
        Returns the current value of the spinbox.

    set(value)
        Sets the value of the spinbox to value.
```
Notebook

Tk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.</td>
</tr>
<tr>
<td>padding</td>
<td>Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.</td>
</tr>
<tr>
<td>width</td>
<td>If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.</td>
</tr>
</tbody>
</table>

Tab Options

There are also specific options for tabs:
<table>
<thead>
<tr>
<th><strong>state</strong></th>
<th>Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sticky</strong></td>
<td>Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the grid() geometry manager.</td>
</tr>
<tr>
<td><strong>padding</strong></td>
<td>Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.</td>
</tr>
<tr>
<td><strong>text</strong></td>
<td>Specifies a text to be displayed in the tab.</td>
</tr>
<tr>
<td><strong>image</strong></td>
<td>Specifies an image to display in the tab. See the option image described in Widget.</td>
</tr>
<tr>
<td><strong>compound</strong></td>
<td>Specifies how to display the image relative to the text, in the case both options text and image are present. See Label Options for legal values.</td>
</tr>
<tr>
<td><strong>underline</strong></td>
<td>Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if Notebook.enable_traversal() is called.</td>
</tr>
</tbody>
</table>

**Tab Identifiers**

The tab_id present in several methods of ttk.Notebook may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the
tab
• The literal string “current”, which identifies the currently-selected tab
• The literal string “end”, which returns the number of tabs (only valid for Notebook.index())

Virtual Events

This widget generates a \texttt{<<NotebookTabChanged>>} virtual event after a new tab is selected.

ttk.Notebook

class \texttt{tkinter.ttk.Notebook}:
\begin{verbatim}
    add(child, **kw)
    Adds a new tab to the notebook.

    If window is currently managed by the notebook but hidden, it is restored to its previous position.

    See Tab Options for the list of available options.

    forget(tab_id)
    Removes the tab specified by \texttt{tab_id}, unmaps and unmanages the associated window.

    hide(tab_id)
    Hides the tab specified by \texttt{tab_id}.

    The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the \texttt{add()} function.
\end{verbatim}
command.

**identify**(x, y)

Returns the name of the tab element at position x, y, or the empty string if none.

**index**(tab_id)

Returns the numeric index of the tab specified by tab_id, or the total number of tabs if tab_id is the string “end”.

**insert**(pos, child, **kw)

 Inserts a pane at the specified position.

pos is either the string “end”, an integer index, or the name of a managed child. If child is already managed by the notebook, moves it to the specified position.

See Tab Options for the list of available options.

**select**(tab_id=None)

Selects the specified tab_id.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If tab_id is omitted, returns the widget name of the currently selected pane.

**tab**(tab_id, option=None, **kw)

Query or modify the options of the specific tab_id.

If kw is not given, returns a dictionary of the tab option values. If option is specified, returns the value of that option. Otherwise, sets the options to the corresponding values.
tabs()
Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- **Control-Tab**: selects the tab following the currently selected one.
- **Shift-Control-Tab**: selects the tab preceding the currently selected one.
- **Alt-K**: where $K$ is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.
Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the progress bar.</td>
</tr>
<tr>
<td>length</td>
<td>Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).</td>
</tr>
<tr>
<td>mode</td>
<td>One of “determinate” or “indeterminate”.</td>
</tr>
<tr>
<td>maximum</td>
<td>A number specifying the maximum value. Defaults to 100.</td>
</tr>
<tr>
<td>value</td>
<td>The current value of the progress bar. In “determinate” mode, this represents the amount of work completed. In “indeterminate” mode, it is interpreted as modulo <code>maximum</code>; that is, the progress bar completes one “cycle” when its value increases by <code>maximum</code>.</td>
</tr>
<tr>
<td>variable</td>
<td>A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.</td>
</tr>
</tbody>
</table>
### phase

Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

### `ttk.Progressbar`

**class** `tkinter.ttk.Progressbar`

**start** *(interval=\textit{None})*

Begin autoincrement mode: schedules a recurring timer event that calls `Progressbar.step()` every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

**step** *(amount=\textit{None})*

Increments the progress bar's value by *amount*.

*amount* defaults to 1.0 if omitted.

**stop**

Stop autoincrement mode: cancels any recurring timer event initiated by `Progressbar.start()` for this progress bar.
Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the separator.</td>
</tr>
</tbody>
</table>
Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a Sizegrip is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ....), the Sizegrip widget will not resize the window.
- This widget supports only “southeast” resizing.
Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option `columns`. See Column Identifiers.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in Scrollable Widget Options and the methods `Treeview.xview()` and `Treeview.yview()`.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>A list of column identifiers, specifying the</td>
</tr>
<tr>
<td><strong>displaycolumns</strong></td>
<td>A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string “#all”.</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>height</strong></td>
<td>Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.</td>
</tr>
<tr>
<td><strong>padding</strong></td>
<td>Specifies the internal padding for the widget. The padding is a list of up to four length specifications.</td>
</tr>
<tr>
<td><strong>selectmode</strong></td>
<td>Controls how the built-in class bindings manage the selection. One of “extended”, “browse” or “none”. If set to “extended” (the default), multiple items may be selected. If “browse”, only a single item will be selected at a time. If “none”, the selection will not be changed.</td>
</tr>
<tr>
<td></td>
<td>Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.</td>
</tr>
<tr>
<td><strong>show</strong></td>
<td>A list containing zero or more of the following values, specifying which elements of the tree to display.</td>
</tr>
<tr>
<td></td>
<td>- tree: display tree labels in column #0.</td>
</tr>
<tr>
<td></td>
<td>- headings: display the heading row.</td>
</tr>
</tbody>
</table>
|                    | The default is “tree headings”, i.e., show all
elements.

**Note:** Column #0 always refers to the tree column, even if `show="tree"` is not specified.

## Item Options

The following item options may be specified for items in the insert and item widget commands.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The textual label to display for the item.</td>
</tr>
<tr>
<td>image</td>
<td>A Tk Image, displayed to the left of the label.</td>
</tr>
<tr>
<td>values</td>
<td>The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.</td>
</tr>
<tr>
<td>open</td>
<td>True/False value indicating whether the item’s children should be displayed or hidden.</td>
</tr>
<tr>
<td>tags</td>
<td>A list of tags associated with this item.</td>
</tr>
</tbody>
</table>

## Tag Options

The following options may be specified on tags:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>foreground</td>
<td>Specifies the text foreground color.</td>
</tr>
<tr>
<td>background</td>
<td>Specifies the cell or item background color.</td>
</tr>
<tr>
<td>font</td>
<td>Specifies the font to use when drawing text.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies the item image, in case the item’s image option is empty.</td>
</tr>
</tbody>
</table>

### Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

### Notes:

- Item’s option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show=”tree” is not specified.

A data column number is an index into an item’s option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

### Virtual Events

The Treeview widget generates the following virtual events.
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;TreeviewSelect&gt;&gt;</td>
<td>Generated whenever the selection changes.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewOpen&gt;&gt;</td>
<td>Generated just before settings the focus item to open=True.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewClose&gt;&gt;</td>
<td>Generated just after setting the focus item to open=False.</td>
</tr>
</tbody>
</table>

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

**ttk.Treeview**

```python
class tkinter.ttk.Treeview

bbox(item, column=None)

Returns the bounding box (relative to the treeview widget's window) of the specified `item` in the form `(x, y, width, height)`. If `column` is specified, returns the bounding box of that cell. If the `item` is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(item=None)

Returns the list of children belonging to `item`. If `item` is not specified, returns root children.

set_children(item, *newchildren)

Replaces `item`'s child with `newchildren`. Children present in `item` that are not present in `newchildren` are detached from the tree. No items in `newchildren` may be an
ancestor of item. Note that not specifying newchildren results in detaching item's children.

**column**(column, option=None, **kw)
Query or modify the options for the specified column.

If kw is not given, returns a dict of the column option values. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id**
  Returns the column name. This is a read-only option.

- **anchor**: One of the standard Tk anchor values.
  Specifies how the text in this column should be aligned with respect to the cell.

- **minwidth**: width
  The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.

- **stretch**: True/False
  Specifies whether the column’s width should be adjusted when the widget is resized.

- **width**: width
  The width of the column in pixels.

To configure the tree column, call this with column = “#0”

**delete**(*items*)
Delete all specified items and all their descendants.
The root item may not be deleted.

**detach(** *items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

**exists(** *item*)
Returns True if the specified *item* is present in the tree.

**focus(** *item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

**heading(** *column*, *option=None*, **kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text:** text
  The text to display in the column heading.

- **image:** imageName
  Specifies an image to display to the right of the column heading.

- **anchor:** anchor
Specifies how the heading text should be aligned. One of the standard Tk anchor values.

- command: callback
  A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with `column = "#0"`.

**identify**(component, x, y)
Returns a description of the specified `component` under the point given by x and y, or the empty string if no such `component` is present at that position.

**identify_row**(y)
Returns the item ID of the item at position y.

**identify_column**(x)
Returns the data column identifier of the cell at position x.

The tree column has ID #0.

**identify_region**(x, y)
Returns one of:

<table>
<thead>
<tr>
<th>region</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>heading</td>
<td>Tree heading area.</td>
</tr>
<tr>
<td>separator</td>
<td>Space between two columns headings.</td>
</tr>
<tr>
<td>tree</td>
<td>The tree area.</td>
</tr>
<tr>
<td>cell</td>
<td>A data cell.</td>
</tr>
</tbody>
</table>

Availability: Tk 8.6.
**identify_element**\((x, y)\)

Returns the element at position \(x, y\).

Availability: Tk 8.6.

**index**\((item)\)

Returns the integer index of \(item\) within its parent's list of children.

**insert**\((parent, index, iid=None, **kw)\)

Creates a new item and returns the item identifier of the newly created item.

\(parent\) is the item ID of the parent item, or the empty string to create a new top-level item. \(index\) is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If \(index\) is less than or equal to zero, the new node is inserted at the beginning; if \(index\) is greater than or equal to the current number of children, it is inserted at the end. If \(iid\) is specified, it is used as the item identifier; \(iid\) must not already exist in the tree. Otherwise, a new unique identifier is generated.

See [Item Options](#) for the list of available points.

**item**\((item, option=None, **kw)\)

Query or modify the options for the specified \(item\).

If no options are given, a dict with options/values for the item is returned. If \(option\) is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by \(kw\).

**move**\((item, parent, index)\)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

**next**(item)

Returns the identifier of *item*'s next sibling, or " if *item* is the last child of its parent.

**parent**(item)

Returns the ID of the parent of *item*, or " if *item* is at the top level of the hierarchy.

**prev**(item)

Returns the identifier of *item*'s previous sibling, or " if *item* is the first child of its parent.

**reattach**(item, parent, index)

An alias for *Treeview.move()*.

**see**(item)

Ensure that *item* is visible.

Sets all of *item*’s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

**selection**(selop=None, items=None)

If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.
Deprecated since version 3.6, will be removed in version 3.8: Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

**selection_set(*items)**

`items` becomes the new selection.

*Changed in version 3.6:* `items` can be passed as separate arguments, not just as a single tuple.

**selection_add(*items)**

Add `items` to the selection.

*Changed in version 3.6:* `items` can be passed as separate arguments, not just as a single tuple.

**selection_remove(*items)**

Remove `items` from the selection.

*Changed in version 3.6:* `items` can be passed as separate arguments, not just as a single tuple.

**selection_toggle(*items)**

Toggle the selection state of each item in `items`.

*Changed in version 3.6:* `items` can be passed as separate arguments, not just as a single tuple.

**set(item, column=None, value=None)**

With one argument, returns a dictionary of column/value pairs for the specified `item`. With two arguments, returns the current value of the specified `column`. With three arguments, sets the value of given `column` in given `item` to the specified `value`. 
**tag_bind**(*tagname*, *sequence=None*, *callback=None*)

Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

**tag_configure**(*tagname*, *option=None*, **kw*)

Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

**tag_has**(*tagname*, *item=None*)

If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

**xview**(*args*)

Query or modify horizontal position of the treeview.

**yview**(*args*)

Query or modify vertical position of the treeview.
Ttk Styling

Each widget in ttk is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method Misc.winfo_class() (somewidget.winfo_class()).

See also:

Tcl'2004 conference presentation
This document explains how the theme engine works

class tkinter.ttk.Style
This class is used to manipulate the style database.

configure(style, query_opt=None, **kw)
Query or set the default value of the specified option(s) in style.

Each key in kw is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```python
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```
```python
ttk.Style().configure("TButton", padding=6, relief=background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

**map(style, query_opt=None, **kw)**

Query or sets dynamic values of the specified option(s) in `style`.

Each key in `kw` is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```python
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=(("pressed", "red"), ("active", ),
    background=(("pressed", "!disabled", "black")

colored_btn = ttk.Button(text="Test", style="C.

root.mainloop()
```

Note that the order of the (states, value) sequences for an
option does matter, if the order is changed to `[(active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

**lookup(style, option, state=None, default=None)**

Returns the value specified for `option` in `style`. If `state` is specified, it is expected to be a sequence of one or more states. If the `default` argument is set, it is used as a fallback value in case no specification for `option` is found.

To check what font a Button uses by default:

```python
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

**layout(style, layoutspec=None)**

Define the widget layout for given `style`. If `layoutspec` is omitted, return the layout specification for given `style`.

`layoutspec`, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in `Layouts`.

To understand the format, see the following example (it is not intended to do anything useful):

```python
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```
element_create(elementname, etype, *args, **kw)

Create a new element in the current theme, of the given etype which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, args should contain the default image name followed by statespec/value pairs (this is the imagespec), and kw may have the following options:

- border=padding
  padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

- height=height
  Specifies a minimum height for the element. If less than zero, the base image’s height is used as a default.
• padding=padding
  Specifies the element’s interior padding. Defaults to border’s value if not specified.

• sticky=spec
  Specifies how the image is placed within the final parcel. spec contains zero or more characters “n”, “s”, “w”, or “e”.

• width=width
  Specifies a minimum width for the element. If less than zero, the base image’s width is used as a default.

If “from” is used as the value of etype, element_create() will clone an existing element. args is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. kw is discarded.

element_names()
  Returns the list of elements defined in the current theme.

element_options(elementname)
  Returns the list of elementname’s options.

theme_create(themename, parent=None, settings=None)
  Create a new theme.

  It is an error if themename already exists. If parent is specified, the new theme will inherit styles, elements and layouts from the parent theme. If settings are present they are expected to have the same syntax used for theme_settings().
**theme_settings**(themename, settings)

Temporarily sets the current theme to `themename`, apply specified `settings` and then restore the previous theme.

Each key in `settings` is a style and each value may contain the keys ‘configure’, ‘map’, ‘layout’ and ‘element create’ and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```python
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
    "TCombobox": {
        "configure": {"padding": 5},
        "map": {
            "background": ["active", "green2"],
            "!disabled": "green4"
        }
    }
})

combo = ttk.Combobox().pack()
```
theme_names()
Returns a list of all known themes.

theme_use(themename=None)
If themename is not given, returns the theme in use. Otherwise, sets the current theme to themename, refreshes all widgets and emits a <<ThemeChanged>> event.

Layouts

A layout can be just None, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- side: whichside
  Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.

- sticky: nswe
  Specifies where the element is placed inside its allocated parcel.

- unit: 0 or 1
  If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of Widget.identify() et al. It's used for things like scrollbar thumbs with grips.

- children: [sublayout... ]
Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a Layout.
tkinter.tix — Extension widgets for Tk

Source code: Lib/tkinter/tix.py

Deprecated since version 3.6: This Tk extension is unmaintained and should not be used in new code. Use tkinter.ttk instead.

The tkinter.tix (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The tkinter.tix library provides most of the commonly needed widgets that are missing from standard Tk: HList, ComboBox, Control (a.k.a. SpinBox) and an assortment of scrollable widgets. tkinter.tix also includes many more widgets that are generally useful in a wide range of applications: NoteBook, FileEntry, PanedWindow, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

See also:

Tix Homepage
The home page for Tix. This includes links to additional documentation and downloads.
**Tix Man Pages**
On-line version of the man pages and reference material.

**Tix Programming Guide**
On-line version of the programmer’s reference material.

**Tix Development Applications**
Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.
Using Tix

class tkinter.tix.Tk(screenName=None, baseName=None, className='Tix')

Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the tkinter.tix module subclasses the classes in the tkinter. The former imports the latter, so to use tkinter.tix with Tkinter, all you need to do is to import one module. In general, you can just import tkinter.tix, and replace the toplevel call to tkinter.Tk with tix.Tk:

```python
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use tkinter.tix, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```python
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```
Tix Widgets

Tix introduces over 40 widget classes to the tkinter repertoire.

Basic Widgets

class tkinter.tix.Balloon

A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.ButtonBox

The ButtonBox widget creates a box of buttons, such as is commonly used for Ok Cancel.

class tkinter.tix.ComboBox

The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.Control

The Control widget is also known as the SpinBox widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.LabelEntry

The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of “entry-form” type of interface.
class tkinter.tix.LabelFrame

The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the frame subwidget and manage them inside the frame subwidget.

class tkinter.tix.Meter

The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.OptionMenu

The OptionMenu creates a menu button of options.

class tkinter.tix.PopupMenu

The PopupMenu widget can be used as a replacement of the tk_popup command. The advantage of the Tix PopupMenu widget is it requires less application code to manipulate.

class tkinter.tix.Select

The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.StdButtonBox

The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class tkinter.tix.DirList

The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the
directories displayed in the list or change to another directory.

class tkinter.tix.DirTree

The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.DirSelectDialog

The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class tkinter.tix.DirSelectBox

The DirSelectBox is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.ExFileSelectBox

The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the ExFileSelectBox widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.FileSelectBox

The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.FileEntry

The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the
button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class tkinter.tix.HList

The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.CheckList

The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.Tree

The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class tkinter.tix.TList

The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.
Manager Widgets

class tkinter.tix.PanedWindow

The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.ListNoteBook

The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the hlist subwidget.

class tkinter.tix.NoteBook

The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual “tabs” at the top of the NoteBook widget.

Image Types

The tkinter.tix module adds:

- pixmap capabilities to all tkinter.tix and tkinter widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from
left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk Button widget.

Miscellaneous Widgets

class tkinter.tix.InputOnly

The InputOnly widgets are to accept inputs from the user, which can be done with the bind command (Unix only).

Form Geometry Manager

In addition, tkinter.tix augments tkinter by providing:

class tkinter.tix.Form

The Form geometry manager based on attachment rules for all Tk widgets.
Tix Commands

class tkinter.tix.tixCommand

The tix commands provide access to miscellaneous elements of Tix’s internal state and the Tix application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```python
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure(cnf=None, **kw)

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.tix_cget(option)

Returns the current value of the configuration option given by option. Option may be any of the configuration options.

tixCommand.tix_getbitmap(name)

Locates a bitmap file of the name name.xpm or name in one of the
bitmap directories (see the \texttt{tix_addbitmapdir()} method). By using \texttt{tix_getbitmap()}, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character @. The returned value can be used to configure the bitmap option of the Tk and Tix widgets.

tixCommand.\texttt{tix_addbitmapdir(directory)}
Tix maintains a list of directories under which the \texttt{tix_getimage()} and \texttt{tix_getbitmap()} methods will search for image files. The standard bitmap directory is \$TIX\_LIBRARY/bitmaps. The \texttt{tix_addbitmapdir()} method adds \texttt{directory} into this list. By using this method, the image files of an applications can also be located using the \texttt{tix_getimage()} or \texttt{tix_getbitmap()} method.

tixCommand.\texttt{tix_filedialog([dlgclass])}
Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to \texttt{tix_filedialog()}. An optional \texttt{dlgclass} parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are \texttt{tix}, \texttt{FileSelectDialog} or \texttt{tixExFileSelectDialog}.

tixCommand.\texttt{tix_getimage(self, name)}
Locates an image file of the name \texttt{name.xpm}, \texttt{name.xbm} or \texttt{name.ppm} in one of the bitmap directories (see the \texttt{tix_addbitmapdir()} method above). If more than one file with the same name (but different extensions) exist, then the image type
is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using `tix_getimage()`, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

```
tixCommand.tix_option_get(name)
```

Gets the options maintained by the Tix scheme mechanism.

```
tixCommand.tix_resetoptions(newScheme, newFontSet[, newScmPrio])
```

Resets the scheme and fontset of the Tix application to `newScheme` and `newFontSet`, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter `newScmPrio` can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and initied, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the `tix_resetoptions()` method must be used.
tkinter.scrolledtext —
Scrolled Text Widget

Source code: Lib/tkinter/scrolledtext.py

The tkinter.scrolledtext module provides a class of the same name which implements a basic text widget which has a vertical scrollbar configured to do the “right thing.” Using the ScrolledText class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the tkinter.Text class.

The text widget and scrollbar are packed together in a Frame, and the methods of the Grid and Pack geometry managers are acquired from the Frame object. This allows the ScrolledText widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

ScrolledText.frame
   The frame which surrounds the text and scroll bar widgets.

ScrolledText.vbar
   The scroll bar widget.
IDLE

Source code: Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment. IDLE has the following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs
Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu (Shell and Editor)

New File
  Create a new file editing window.

Open…
  Open an existing file with an Open dialog.

Recent Files
  Open a list of recent files. Click one to open it.

Open Module…
  Open an existing module (searches sys.path).

Class Browser
  Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As…
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As…
Save the current window to different file without changing the associated file.

Print Window
Print the current window to the default printer.

Close
Close the current window (ask to save if unsaved).

Exit
Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Undo
Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo
Redo the last undone change to the current window.

Cut
Copy selection into the system-wide clipboard; then delete the selection.
Copy
  Copy selection into the system-wide clipboard.

Paste
  Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All
  Select the entire contents of the current window.

Find…
  Open a search dialog with many options

Find Again
  Repeat the last search, if there is one.

Find Selection
  Search for the currently selected string, if there is one.

Find in Files…
  Open a file search dialog. Put results in a new output window.

Replace…
  Open a search-and-replace dialog.

Go to Line
  Move cursor to the line number requested and make that line visible.

Show Completions
  Open a scrollable list allowing selection of keywords and attributes.
  See Completions in the Editing and navigation section below.

Expand Word
  Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip
After an unclosed parenthesis for a function, open a small window with function parameter hints. See Calltips in the Editing and navigation section below.

Show surrounding parens
   Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region
   Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
   Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
   Insert ## in front of selected lines.

Uncomment Region
   Remove leading # or ## from selected lines.

Tabify Region
   Turn leading stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
   Turn all tabs into the correct number of spaces.

Toggle Tabs
   Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
   Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph
   Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N
defaults to 72.

Strip trailing whitespace

Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying `str.rstrip` to each line, including lines within multiline strings.

Run menu (Editor window only)

Python Shell

Open or wake up the Python Shell window.

Check Module

Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Run Module

Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Shell menu (Shell window only)

View Last Restart

Scroll the shell window to the last Shell restart.

Restart Shell

Restart the shell to clean the environment.
Previous History
   Cycle through earlier commands in history which match the current entry.

Next History
   Cycle through later commands in history which match the current entry.

Interrupt Execution
   Stop a running program.

Debug menu (Shell window only)

Go to File/Line
   Look on the current line, with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
   When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
   Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
   Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)
Configure IDLE

Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more, see Setting preferences under Help and preferences.

Zoom/Restore Height

Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

Show/Hide Code Context (Editor Window only)

Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See Code Context in the Editing and Navigation section below.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE

Display version, copyright, license, credits, and more.

IDLE Help

Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs

Access local Python documentation, if installed, or start a web
browser and open docs.python.org showing the latest Python documentation.

Turtle Demo

Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the Help sources subsection below for more on Help menu choices.

Context Menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut

Copy selection into the system-wide clipboard; then delete the selection.

Copy

Copy selection into the system-wide clipboard.

Paste

Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

Set Breakpoint

Set a breakpoint on the current line.
Clear Breakpoint
  Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line
  Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the the Python Shell window subsection below.

Squeeze
  If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a ‘Squeezed text’ label.
Editing and navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number (‘Ln’) and column number (‘Col’). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

In this section, ‘C’ refers to the Control key on Windows and Unix and the Command key on macOS.

- Backspace deletes to the left; Del deletes to the right
- C-Backspace delete word left; C-Del delete word to the right
- Arrow keys and Page Up/Page Down to move around
- C-LeftArrow and C-RightArrow moves by words
- Home/End go to begin/end of line
- C-Home/C-End go to begin/end of file
• Some useful Emacs bindings are inherited from Tcl/Tk:
  
  - c-a beginning of line
  - c-e end of line
  - c-k kill line (but doesn’t put it in clipboard)
  - c-l center window around the insertion point
  - c-b go backward one character without deleting (usually you can also use the cursor key for this)
  - c-f go forward one character without deleting (usually you can also use the cursor key for this)
  - c-p go up one line (usually you can also use the cursor key for this)
  - c-d delete next character

  Standard keybindings (like c-c to copy and c-v to paste) may work. Keybindings are selected in the Configure IDLE dialog.

**Automatic indentation**

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the Format menu.

**Completions**

Completions are supplied for functions, classes, and attributes of
classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a ‘.’ or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a Tab will supply that completion without opening the ACW.

‘Show Completions’ will force open a completions window, by default the C-space will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a tab will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two tab in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

“Hidden” attributes can be accessed by typing the beginning of hidden name after a ‘.’, e.g. ‘_’. This allows access to modules with __all__ set, or to class-private attributes.

Completions and the ‘Expand Word’ facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names
in an Editor window which are not via __main__ and sys.modules will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don’t like the ACW popping up unbidden, simply make the delay longer or disable the extension.

Calltips

A calltip is shown when one types ( after the name of an accessible function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or ) is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter itertools.count(). A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter turtle.write( and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter import turtle and then turtle.write( will work.

In an editor, import statements have no effect until one runs the file.
One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

**Code Context**

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

**Python Shell window**

With IDLE’s Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a `SyntaxError` when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when
entering code interactively. IDLE’s Shell window also responds to the following keys.

- **C-c** interrupts executing command
- **C-d** sends end-of-file; closes window if typed at a >>> prompt
- **Alt-/** (Expand word) is also useful to reduce typing

**Command history**

- **Alt-p** retrieves previous command matching what you have typed. On macOS use **c-p**.
- **Alt-n** retrieves next. On macOS use **c-n**.
- **Return** while on any previous command retrieves that command

**Text colors**

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.
Startup and code execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables IDLESTARTUP or PYTHONSTARTUP. IDLE first checks for IDLESTARTUP; if IDLESTARTUP is present the file referenced is run. If IDLESTARTUP is not present, IDLE checks for PYTHONSTARTUP. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, Tk also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `Idle.py` and is looked for in the user’s home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE’s Python shell.

Command line usage

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [arg] ...
```

-c command  run command in the shell window
-d           enable debugger and open shell window
-e           open editor window
-h           print help message with legal combinations
-i           open shell window
-r file      run file in shell window
-s           run $IDLESTARTUP or $PYTHONSTARTUP first
-t title     set title of shell window
-             run stdin in shell (- must be last option)
```
If there are arguments:

- If `-`, `-c`, or `-r` is used, all arguments are placed in `sys.argv[1:]` and `sys.argv[0]` is set to `'`, `-c`, or `-r`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

**Startup failure**

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says ‘RESTART’). If the user process fails to connect to the GUI process, it displays a Tk error box with a ‘cannot connect’ message that directs the user here. It then exits.

A common cause of failure is a user-written file with the same name as a standard library module, such as `random.py` and `tkinter.py`. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one
undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to detect and stop one. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in ~/.idlerc/ (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand, using the configuration dialog, under Options, instead Options. Once it happens, the solution may be to delete one or more of the configuration files.

If IDLE quits with no message, and it was not started from a console, try starting from a console (python -m idlelib) and see if a message appears.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, sys.modules starts with more entries, and threading.activeCount() returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces sys.stdin, sys.stdout, and
sys.stderr with objects that get input from and send output to the Shell window. The original values stored in sys.__stdin__, sys.__stdout__, and sys.__stderr__ are not touched, but may be None.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE’s standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use input from sys.stdin or print or write to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

If sys is reset by user code, such as with importlib.reload(sys), IDLE’s changes are lost and input from the keyboard and output to the screen will not work correctly.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, sys.stdout and sys.stderr are connected to the display area of IDLE’s Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends
unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE’s Shell, displays characters (codepoints) in the the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 ‘characters’). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```python
>>> s = 'a\tb\a<x02><r>b\n\nd'
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<x02><r>\x08c\n\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font. Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight
colors.

For SyntaxError tracebacks, the normal ‘^’ marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right-click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a ‘Squeezed text’ label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does
not return until the tk app is destroyed. If the program is run with
```
python -i
```
or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

## Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the `-n` command line switch.

If IDLE is started with the `-n` command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import
baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

*Deprecated since version 3.4.*
Help and preferences

Help sources

Help menu entry “IDLE Help” displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry “Python Docs” opens the extensive sources of help, including tutorials, available at docs.python.org/x.y, where ‘x.y’ is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a .idlerc directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or
an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

**IDLE on macOS**

Under System Preferences: Dock, one can set “Prefer tabs when opening documents” to “Always”. This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

**Extensions**

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.
Other Graphical User Interface Packages

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:

See also:

**PyGObject**
PyGObject provides introspection bindings for C libraries using GObject. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online [PyGObject](https://pygobject.readthedocs.io) tutorial is available.

**PyGTK**
PyGTK provides bindings for an older version of the library, GTK+ 2. It provides an object oriented interface that is slightly higher level than the C one. There are also bindings to GNOME. An online [tutorial](https://python-gtk3-demo.github.io/) is available.

**PyQt**
PyQt is a [sip](https://www.sip-tool.org)-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. sip is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python.

**PySide**
PySide is a newer binding to the Qt toolkit, provided by Nokia. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

**wxPython**
wxPython is a cross-platform GUI toolkit for Python that is built
around the popular **wxWidgets** (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform’s native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules.

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the [GUI Programming](https://wiki.python.org/moin/GUIProgramming) page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.
Development Tools

The modules described in this chapter help you write software. For example, the `pydoc` module takes a module and generates documentation based on the module’s contents. The `doctest` and `unittest` modules contain frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced. `2to3` can translate Python 2.x source code into valid Python 3.x code.

The list of modules described in this chapter is:

- **typing** — Support for type hints
  - Type aliases
  - `NewType`
  - Callable
  - Generics
  - User-defined generic types
  - The Any type
  - Classes, functions, and decorators
- **pydoc** — Documentation generator and online help system
- **doctest** — Test interactive Python examples
  - Simple Usage: Checking Examples in Docstrings
  - Simple Usage: Checking Examples in a Text File
  - How It Works
    - Which Docstrings Are Examined?
    - How are Docstring Examples Recognized?
    - What’s the Execution Context?
    - What About Exceptions?
    - Option Flags
    - Directives
    - Warnings
• Basic API
• Unittest API
• Advanced API
  ■ DocTest Objects
  ■ Example Objects
  ■ DocTestFinder objects
  ■ DocTestParser objects
  ■ DocTestRunner objects
  ■ OutputChecker objects
• Debugging
• Soapbox
• unittest — Unit testing framework
  ○ Basic example
  ○ Command-Line Interface
    ■ Command-line options
  ○ Test Discovery
  ○ Organizing test code
  ○ Re-using old test code
  ○ Skipping tests and expected failures
  ○ Distinguishing test iterations using subtests
  ○ Classes and functions
    ■ Test cases
      ■ Deprecated aliases
    ■ Grouping tests
    ■ Loading and running tests
      ■ load_tests Protocol
  ○ Class and Module Fixtures
    ■ setUpClass and tearDownClass
    ■ setUpModule and tearDownModule
  ○ Signal Handling
• unittest.mock — mock object library
  ○ Quick Guide
  ○ The Mock Class
    ■ Calling
- Deleting Attributes
  - Mock names and the name attribute
  - Attaching Mocks as Attributes
- The patchers
  - patch
  - patch.object
  - patch.dict
  - patch.multiple
  - patch methods: start and stop
  - patch builtins
  - TEST_PREFIX
- Nesting Patch Decorators
- Where to patch
- Patching Descriptors and Proxy Objects
- MagicMock and magic method support
  - Mocking Magic Methods
  - Magic Mock
- Helpers
  - sentinel
  - DEFAULT
  - call
  - create_autospec
  - ANY
  - FILTER_DIR
  - mock_open
  - Autospecing
  - Sealing mocks
- `unittest.mock` — getting started
  - Using Mock
    - Mock Patching Methods
    - Mock for Method Calls on an Object
    - Mocking Classes
    - Naming your mocks
    - Tracking all Calls
- Setting Return Values and Attributes
- Raising exceptions with mocks
- Side effect functions and iterables
- Creating a Mock from an Existing Object
  - Patch Decorators
  - Further Examples
    - Mocking chained calls
    - Partial mocking
    - Mocking a Generator Method
    - Applying the same patch to every test method
    - Mocking Unbound Methods
    - Checking multiple calls with mock
    - Coping with mutable arguments
    - Nesting Patches
    - Mocking a dictionary with MagicMock
    - Mock subclasses and their attributes
    - Mocking imports with patch.dict
    - Tracking order of calls and less verbose call assertions
    - More complex argument matching

- 2to3 - Automated Python 2 to 3 code translation
  - Using 2to3
  - Fixers
  - lib2to3 - 2to3's library

- test — Regression tests package for Python
  - Writing Unit Tests for the test package
  - Running tests using the command-line interface

- test.support — Utilities for the Python test suite

- test.support.script_helper — Utilities for the Python execution tests

See also the Python development mode: the -X dev option and PYTHONDEVMODE environment variable.
typing — Support for type hints

New in version 3.5.

Source code: Lib/typing.py

Note: The typing module has been included in the standard library on a provisional basis. New features might be added and API may change even between minor releases if deemed necessary by the core developers.

This module supports type hints as specified by PEP 484 and PEP 526. The most fundamental support consists of the types Any, Union, Tuple, Callable, TypeVar, and Generic. For full specification please see PEP 484. For a simplified introduction to type hints see PEP 483.

The function below takes and returns a string and is annotated as follows:

```python
def greeting(name: str) -> str:
    return 'Hello ' + name
```

In the function `greeting`, the argument `name` is expected to be of type `str` and the return type `str`. Subtypes are accepted as arguments.
Type aliases

A type alias is defined by assigning the type to the alias. In this example, `Vector` and `List[float]` will be treated as interchangeable synonyms:

```python
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example:

```python
from typing import Dict, Tuple, Sequence

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence):
    ...

# The static type checker will treat the previous type
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[Tuple[Tuple[str, int], Dict]],
    ...
```
Note that `None` as a type hint is a special case and is replaced by `type(None)`.
NewType

Use the `NewType()` helper function to create distinct types:

```python
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors:

```python
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a `UserId` in an invalid way:

```python
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime the statement `Derived = NewType('Derived', Base)` will make `Derived` a function that immediately returns whatever parameter you pass it. That means the expression
Derived(some_value) does not create a new class or introduce any overhead beyond that of a regular function call.

More precisely, the expression some_value is Derived(some_value) is always true at runtime.

This also means that it is not possible to create a subtype of Derived since it is an identity function at runtime, not an actual type:

```python
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

However, it is possible to create a NewType() based on a ‘derived’ NewType:

```python
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

and typechecking for ProUserId will work as expected.

See PEP 484 for more details.

**Note:** Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing Alias = Original will make the static type checker treat Alias as being *exactly equivalent* to Original in all cases. This is useful when you want to simplify complex type signatures.
In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

*New in version 3.5.2.*
Callable

Frameworks expecting callback functions of specific signatures might be type hinted using `Callable[[Arg1Type, Arg2Type], ReturnType]`.

For example:

```python
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                 on_error: Callable[[int, Exception],
                                    None],
                 # Body
```

It is possible to declare the return type of a callable without specifying the call signature by substituting a literal ellipsis for the list of arguments in the type hint: `Callable[..., ReturnType]`. 
Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, abstract base classes have been extended to support subscription to denote expected types for container elements.

```python
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                      overrides: Mapping[str, str]) -> 
```

Generics can be parameterized by using a new factory available in typing called `TypeVar`.

```python
from typing import Sequence, TypeVar

T = TypeVar('T') # Declare type variable

def first(l: Sequence[T]) -> T: # Generic function
    return l[0]
```
User-defined generic types

A user-defined class can be defined as a generic class.

```python
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger):
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

Generic[T] as a base class defines that the class LoggedVar takes a single type parameter T. This also makes T valid as a type within the class body.

The Generic base class uses a metaclass that defines __getitem__() so that LoggedVar[t] is valid as a type:

```python
from typing import Iterable
```
A generic type can have any number of type variables, and type variables may be constrained:

```python
def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

Each type variable argument to `Generic` must be distinct. This is thus invalid:

```python
from typing import TypeVar, Generic

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

You can use multiple inheritance with `Generic`:

```python
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```
When inheriting from generic classes, some type variables could be fixed:

```python
from typing import TypeVar, Mapping
T = TypeVar('T')
class MyDict(Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes `Any` for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```python
from typing import Iterable

class MyIterable(Iterable):  # Same as Iterable[Any]
    ...
```

User defined generic type aliases are also supported. Examples:

```python
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]
```
```python
def inproduct(v: Vec[T]) -> T:  # Same as Iterable[T]
    return sum(x*y for x, y in v)
```

The metaclass used by `Generic` is a subclass of `abc.ABCMeta`. A generic class can be an ABC by including abstract methods or properties, and generic classes can also have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are hashable and comparable for equality.
The *Any* type

A special kind of type is *Any*. A static type checker will treat every type as being compatible with *Any* and *Any* as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type on *Any* and assign it to any variable:

```python
from typing import Any

a = None  # type: Any
a = []    # OK
a = 2     # OK

s = ''    # type: str
s = a     # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no typechecking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type *str* and receives an *int* value at runtime!

Furthermore, all functions without a return type or parameter types will implicitly default to using *Any*:

```python
def legacy_parser(text):
```
This behavior allows \texttt{Any} to be used as an \textit{escape hatch} when you need to mix dynamically and statically typed code.

Contrast the behavior of \texttt{Any} with the behavior of \texttt{object}. Similar to \texttt{Any}, every type is a subtype of \texttt{object}. However, unlike \texttt{Any}, the reverse is not true: \texttt{object} is \textit{not} a subtype of every other type.

That means when the type of a value is \texttt{object}, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

\begin{Verbatim}
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of
hash_a(42)
hash_a("foo")
\end{Verbatim}
# Typechecks, since Any is compatible with all types

```python
hash_b(42)
hash_b("foo")
```

Use `object` to indicate that a value could be any type in a typesafe manner. Use `Any` to indicate that a value is dynamically typed.
Classes, functions, and decorators

The module defines the following classes, functions and decorators:

class typing. TypeVar
    Type variable.

Usage:

```python
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or by
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

```python
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x"
    return [x] * n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings.""
    return x if len(x) >= len(y) else y
```

The latter example’s signature is essentially the overloading of (str, str) -> str and (bytes, bytes) -> bytes. Also note that if the arguments are instances of some subclass of str, the return type is still plain str.

At runtime, isinstance(x, T) will raise TypeError. In general, isinstance() and issubclass() should not be used
with types.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See PEP 484 for more details. By default type variables are invariant. Alternatively, a type variable may specify an upper bound using `bound=<type>`. This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subclass of the boundary type, see PEP 484.

```python
class typing.Generic
    Abstract base class for generic types.

    A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as:

    ```python
class Mapping(Generic[KT, VT]):
        def __getitem__(self, key: KT) -> VT:
            ...
            # Etc.
    ```

    This class can then be used as follows:

    ```python
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default):
    try:
        return mapping[key]
    except KeyError:
        return default
    ```
```
```
A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `Type[C]` may accept values that are classes themselves – specifically, it will accept the class `object` of `C`. For example:

```python
a = 3  # Has type 'int'
b = int  # Has type 'Type[int]'
c = type(a)  # Also has type 'Type[int]'
```

Note that `Type[C]` is covariant:

```python
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User
    # ...
    return user_class()
```

The fact that `Type[C]` is covariant implies that all subclasses of `C` should implement the same constructor signature and class method signatures as `C`. The type checker should flag violations of this, but should also allow constructor calls in subclasses that match the constructor calls in the indicated base class. How the type checker is required to handle this particular case may change in future revisions of PEP 484.

The only legal parameters for `Type` are classes, `Any`, `type variables`, and unions of any of these types. For example:

```python
def new_non_team_user(user_class: Type[Union[Base
```
Type[Any] is equivalent to Type which in turn is equivalent to type, which is the root of Python's metaclass hierarchy.

*New in version 3.5.2.*

```python
class typing.Iterable(Generic[T_co])
    A generic version of collections.abc.Iterable.

class typing.Iterator(Iterable[T_co])
    A generic version of collections.abc.Iterator.

class typing.Reversible(Iterable[T_co])
    A generic version of collections.abc.Reversible.

class typing.SupportsInt
    An ABC with one abstract method __int__.

class typing.SupportsFloat
    An ABC with one abstract method __float__.

class typing.SupportsComplex
    An ABC with one abstract method __complex__.

class typing.SupportsBytes
    An ABC with one abstract method __bytes__.

class typing.SupportsAbs
    An ABC with one abstract method __abs__ that is covariant in its return type.

class typing.SupportsRound
    An ABC with one abstract method __round__ that is covariant in
its return type.

```python
class typing.Container[Generic[T_co])
    A generic version of collections.abc.Container.
```

```python
class typing.Hashable
    An alias to collections.abc.Hashable
```

```python
class typing.Sized
    An alias to collections.abc.Sized
```

```python
class typing.Collection(Sized, Iterable[T_co],
                        Container[T_co])
    A generic version of collections.abc.Collection

    New in version 3.6.
```

```python
class typing.AbstractSet(Sized, Collection[T_co])
    A generic version of collections.abc.Set.
```

```python
class typing.MutableSet(AbstractSet[T])
    A generic version of collections.abc.MutableSet.
```

```python
class typing.Mapping(Sized, Collection[KT],
                    Generic[VT_co])
    A generic version of collections.abc.Mapping. This type can
    be used as follows:

    ```python
def get_position_in_index(word_list: Mapping[str,
                               return word_list[word]
```

```python
class typing.MutableMapping(Mapping[KT, VT])
    A generic version of collections.abc.MutableMapping.
```
class typing.Sequence(Reversible[T_co], Collection[T_co])
A generic version of collections.abc.Sequence.

class typing.MutableSequence(Sequence[T])
A generic version of collections.abc.MutableSequence.

class typing.ByteString(Sequence[int])
A generic version of collections.abc.ByteString.

This type represents the types bytes, bytearray, and memoryview.

As a shorthand for this type, bytes can be used to annotate arguments of any of the types mentioned above.

class typing.Deque(deque, MutableSequence[T])
A generic version of collections.deque.

New in version 3.6.1.

class typing.List(list, MutableSequence[T])
Generic version of list. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as Sequence or Iterable.

This type may be used as follows:

```python
T = TypeVar('T', int, float)
def vec2(x: T, y: T) -> List[T]:
    return [x, y]
def keep_positives(vector: Sequence[T]) -> List[T]
    return [item for item in vector if item > 0]
```
class typing.Set(set, MutableSet[T])
   A generic version of builtins.set. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as AbstractSet.

class typing.FrozenSet(frozenset, AbstractSet[T_co])
   A generic version of builtins.frozenset.

class typing.MappingView(Sized, Iterable[T_co])
   A generic version of collections.abc.MappingView.

class typing.KeysView(MappingView[KT_co], AbstractSet[KT_co])
   A generic version of collections.abc.KeysView.

class typing.ItemsView(MappingView, Generic[KT_co, VT_co])
   A generic version of collections.abc.ItemsView.

class typing.ValuesView(MappingView[VT_co])
   A generic version of collections.abc.ValuesView.

class typing.Awaitable(Generic[T_co])
   A generic version of collections.abc.Awaitable.

class typingCoroutine(Awaitable[V_co], Generic[T_co T_contra, V_co])
   A generic version of collections.abcCoroutine. The variance and order of type variables correspond to those of Generator, for example:

```python
from typing import List, Coroutine
```
```python
c = None  # type: Coroutine[List[str], str, int]
...
x = c.send('hi')  # type: List[str]
async def bar() -> None:
    x = await c  # type: int
```

```python
class typing.AsyncIterable(Generic[T_co])
    A generic version of collections.abc.AsyncIterable.

class typing.AsyncIterator(AsyncIterable[T_co])
    A generic version of collections.abc.AsyncIterator.

class typing.ContextManager(Generic[T_co])
    A generic version of contextlib.AbstractContextManager.

    New in version 3.6.

class typing.AsyncContextManager(Generic[T_co])
    A generic version of contextlib.AbstractAsyncContextManager.

    New in version 3.6.

class typing.Dict(dict, MutableMapping[KT, VT])
    A generic version of dict. Useful for annotating return types. To
    annotate arguments it is preferred to use an abstract collection
type such as Mapping.

    This type can be used as follows:

```python
def count_words(text: str) -> Dict[str, int]:
    ...
```

class typing.DefaultDict(collections.defaultdict,
MutableMapping[KT, VT])
    A generic version of collections.defaultdict.

    New in version 3.5.2.

class typing.OrderedDict(collections.OrderedDict, MutableMapping[KT, VT])
    A generic version of collections.OrderedDict.

    New in version 3.7.2.

class typing.Counter(collections.Counter, Dict[T, int])
    A generic version of collections.Counter.

    New in version 3.6.1.

class typing.ChainMap(collections.ChainMap, MutableMapping[KT, VT])
    A generic version of collections.ChainMap.

    New in version 3.6.1.

class typing.Generator(Iterator[T_co], Generic[T_co, T_contra, V_co])
    A generator can be annotated by the generic type Generator[YieldType, SendType, ReturnType]. For example:

    ```python
def echo_round() -> Generator[int, float, str]:
        sent = yield 0
        while sent >= 0:
            sent = yield round(sent)
        return 'Done'
```
Note that unlike many other generics in the typing module, the `SendType` of `Generator` behaves contravariantly, not covariantly or invariantly.

If your generator will only yield values, set the `SendType` and `ReturnType` to `None`:

```python
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator as having a return type of either `Iterable[YieldType]` or `Iterator[YieldType]`:

```python
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class typing. `AsyncGenerator`( `AsyncIterator[T_co]`, `Generic[T_co, T_contra]` )

An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`. For example:

```python
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value,
so there is no `ReturnType` type parameter. As with `Generator`, the `SendType` behaves contravariantly.

If your generator will only yield values, set the `SendType` to `None`:

```python
async def infinite_stream(start: int) -> AsyncGenerator
    while True:
        yield start
        start = await increment(start)
```

Alternatively, annotate your generator as having a return type of either `AsyncIterable[YieldType]` or `AsyncIterator[YieldType]`:

```python
async def infinite_stream(start: int) -> AsyncIterable
    while True:
        yield start
        start = await increment(start)
```

*New in version 3.5.4.*

class typing.Text

`Text` is an alias for `str`. It is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```python
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

*New in version 3.5.2.*
class typing.IO
class typing.TextIO
class typing.BinaryIO

Generic type IO[AnyStr] and its subclasses TextIO(IO[str]) and BinaryIO(IO[bytes]) represent the types of I/O streams such as returned by open().

class typing.Pattern
class typing.Match

These type aliases correspond to the return types from re.compile() and re.match(). These types (and the corresponding functions) are generic in AnyStr and can be made specific by writing Pattern[str], Pattern[bytes], Match[str], or Match[bytes].

class typing.NamedTuple

Typed version of namedtuple.

Usage:

definition:

```python
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```python
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```python
class Employee(NamedTuple):
```
Fields with a default value must come after any fields without a default.

The resulting class has two extra attributes: `_field_types`, giving a dict mapping field names to types, and `_field_defaults`, a dict mapping field names to default values. (The field names are in the `_fields` attribute, which is part of the namedtuple API.)

NamedTuple subclasses can also have docstrings and methods:

```python
class Employee(NamedTuple):
    """Represents an employee.""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>
```

Backward-compatible usage:

```python
Employee = NamedTuple('Employee', [('name', str), ...
```

*Changed in version 3.6:* Added support for PEP 526 variable annotation syntax.

*Changed in version 3.6.1:* Added support for default values,
methods, and docstrings.

typing.**NewType**(typ)
A helper function to indicate a distinct types to a typechecker, see **NewType**. At runtime it returns a function that returns its argument.

Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

*New in version 3.5.2.*

typing.**cast**(typ, val)
Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don’t check anything (we want this to be as fast as possible).

typing.**get_type_hints**(obj[, globals[, locals]])
Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. If necessary, `Optional[t]` is added for function and method annotations if a default value equal to `None` is set. For a class `C`, return a dictionary constructed by merging all the `__annotations__` along `C.__mro__` in reverse order.

@typing.**overload**
The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise `NotImplementedError`. An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```python
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

See PEP 484 for details and comparison with other typing semantics.

`@typing.no_type_check`

Decorator to indicate that annotations are not type hints.

This works as class or function decorator. With a class, it applies recursively to all methods defined in that class (but not to methods
defined in its superclasses or subclasses).

This mutates the function(s) in place.

@typing.no_type_check_decorator
Decorator to give another decorator the no_type_check() effect.

This wraps the decorator with something that wraps the decorated function in no_type_check().

typing.Any
Special type indicating an unconstrained type.

- Every type is compatible with Any.
- Any is compatible with every type.

typing.NoReturn
Special type indicating that a function never returns. For example:

```python
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

New in version 3.5.4.

typing.Union
Union type; Union[X, Y] means either X or Y.

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:
Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually
```

Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

You cannot subclass or instantiate a union.

You cannot write `Union[X][Y]`.

You can use `Optional[X]` as a shorthand for `Union[X, None]`.

*Changed in version 3.7:* Don’t remove explicit subclasses from unions at runtime.

**typing.Optional**

Optional type.

`Optional[X]` is equivalent to `Union[X, None]`.

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:
```python
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of None is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```python
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing. **Tuple**

Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type `X` and the second of type `Y`.

Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables T1 and T2. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain `Tuple` is equivalent to `Tuple[Any, ...]`, and in turn to `tuple`.

typing. **Callable**

Callable type; `Callable[[int], str]` is a function of `(int) -> str`.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types or an ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments; such function types are rarely used as callback types. `Callable[...,
ReturnType] (literal ellipsis) can be used to type hint a callable taking any number of arguments and returning ReturnType. A plain Callable is equivalent to Callable[... , Any], and in turn to collections.abc.Callable.

typing.ClassVar
Special type construct to mark class variables.

As introduced in PEP 526, a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10 # instance variable

ClassVar accepts only types and cannot be further subscribed.

ClassVar is not a class itself, and should not be used with isinstance() or issubclass(). ClassVar does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

typing.AnyStr

New in version 3.5.3.
AnyStr is a type variable defined as `AnyStr = TypeVar('AnyStr', str, bytes)`.

It is meant to be used for functions that may accept any kind of string without allowing different kinds of strings to mix. For example:

```python
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

**typing.TYPE_CHECKING**

A special constant that is assumed to be True by 3rd party static type checkers. It is False at runtime. Usage:

```python
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_
```

Note that the first type annotation must be enclosed in quotes, making it a “forward reference”, to hide the `expensive_mod` reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

*New in version 3.5.2.*
The `pydoc` module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the `__doc__` attribute) of the object, and recursively of its documentable members. If there is no docstring, `pydoc` tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see `inspect.getcomments()`).

The built-in function `help()` invokes the online help system in the interactive interpreter, which uses `pydoc` to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running `pydoc` as a script at the operating system’s command prompt. For example, running

```
pydoc sys
```

at a shell prompt will display documentation on the `sys` module, in a style similar to the manual pages shown by the Unix `man` command. The argument to `pydoc` can be the name of a function, module, or
package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to `pydoc` looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

**Note:** In order to find objects and their documentation, `pydoc` imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, `pydoc` attempts to paginate the output for easier reading. If the PAGER environment variable is set, `pydoc` will use its value as a pagination program.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix `man` command. The synopsis line of a module is the first line of its documentation string.

You can also use `pydoc` to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. `pydoc -p 1234` will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred Web browser. Specifying `0` as the port number will select an arbitrary unused port.
**pydoc** `-n <hostname>` will start the server listening at the given hostname. By default the hostname is `localhost` but if you want the server to be reached from other machines, you may want to change the host name that the server responds to. During development this is especially useful if you want to run **pydoc** from within a container.

**pydoc** `-b` will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can Get help on an individual item, Search all modules with a keyword in their synopsis line, and go to the Module index, Topics and Keywords pages.

When **pydoc** generates documentation, it uses the current environment and path to locate modules. Thus, invoking **pydoc spam** documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in [https://docs.python.org/X.Y/library/](https://docs.python.org/X.Y/library/) where `X` and `Y` are the major and minor version numbers of the Python interpreter. This can be overridden by setting the PYTHONDOCS environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

*Changed in version 3.2:* Added the `-b` option.

*Changed in version 3.3:* The `-g` command line option was removed.

*Changed in version 3.4:* **pydoc** now uses `inspect.signature()` rather than `inspect.getfullargspec()` to extract signature information from callables.

*Changed in version 3.7:* Added the `-n` option.
doctest — Test interactive Python examples

Source code: Lib/doctest.py

The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module’s docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of “literate testing” or “executable documentation”.

Here’s a complete but small example module:

"""
This is the "example" module.

The example module supplies one function, factorial().

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >"""
>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
265252859812191058636308480000000
>>> factorial(-1)
Traceback (most recent call last):
... ValueError: n must be >= 0

Factorials of floats are OK, but the float must
>>> factorial(30.1)
Traceback (most recent call last):
... ValueError: n must be exact integer
>>> factorial(30.0)
265252859812191058636308480000000

It must also not be ridiculously large:
>>> factorial(1e100)
Traceback (most recent call last):
... OverflowError: n too large

"""

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n:  # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

if __name__ == "__main__":
    import doctest
doctest.testmod()

If you run example.py directly from the command line, doctest works its magic:

$ python example.py
$

There’s no output! That’s normal, and it means all the examples worked. Pass -v to the script, and doctest prints a detailed log of what it’s trying, and prints a summary at the end:

$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
  ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
  ok

And so on, eventually ending with:

Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
That's all you need to know to start making productive use of **doctest**! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest.py`. 
Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module M with:

```python
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module M.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```bash
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is **Test Failed** N failures., where N is the number of examples that failed.

Run it with the -v switch instead:

```bash
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to `testmod()`, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by `testmod()` (so passing
There is also a command line shortcut for running `testmod()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run `testmod()` on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on `testmod()`, see section **Basic API**.
Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the `testfile()` function:

```python
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn’t need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
----------------

This is an example text file in reStructuredText for ``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the
error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with `testmod()`, `testfile()` won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.

By default, `testfile()` looks for files in the calling module's directory. See section Basic API for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument `verbose`.

There is also a command line shortcut for running `testfile()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, `doctest` infers that it must be run with `testfile()`, not `testmod()`.

For more information on `testfile()`, see section Basic API.
How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if M.__test__ exists and “is true”, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from M.__test__ are searched, and strings are treated as if they were docstrings. In output, a key K in M.__test__ appears with name

<name of M>.__test__.K

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn’t trying to do an exact emulation of any
specific Python shell.

```python
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
no
NO
NO!!!
```

Any expected output must immediately follow the final `>>>` or `...` line containing the code, and the expected output (if any) extends to the next `>>>` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.

- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output are expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the
NORMALIZE_WHITESPACE option or directive is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom DocTestParser class.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).

- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```python
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m
```

Otherwise, the backslash will be interpreted as part of the string. For example, the \n above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```python
>>> def f(x):
...     '''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m
```

- The starting column doesn’t matter:
>>> assert "Easy!"
    >>> import math
    >>> math.floor(1.9)
    1

and as many leading whitespace characters are stripped from the expected output as appeared in the initial '>>> ' line that started the example.

What’s the Execution Context?

By default, each time doctest finds a docstring to test, it uses a shallow copy of M’s globals, so that running tests doesn’t change the module’s real globals, and so that one test in M can’t leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing globs=your_dict to testmod() or testfile() instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback. [1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:
That doctest succeeds if \texttt{ValueError} is raised, with the \texttt{list.remove(x): x not in list} detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

\begin{verbatim}
Traceback (most recent call last):
Traceback (innermost last):
\end{verbatim}

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

\begin{verbatim}
>>> \texttt{raise ValueError('multi\n line\ndetail')}
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: multi
 line
detail
\end{verbatim}

The last three lines (starting with \texttt{ValueError}) are compared against the exception's type and detail, and the rest are ignored.
Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```python
g>>> raise ValueError('multi
  line\ndetail')
Traceback (most recent call last):
...  
ValueError: multi
  line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's `ELLIPSIS` option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, or start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the `IGNORE_EXCEPTION_DETAIL` doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some SyntaxErrors. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a SyntaxError that omits the traceback header, you will need to manually add the traceback header line to your test example.

- For some SyntaxErrors, Python displays the character position of the syntax error, using a ^ marker:

```
>>> 1 1
   File "<stdin>", line 1
   1 1
   ^
SyntaxError: invalid syntax
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

```
>>> 1 1
   File "<stdin>", line 1
   1 1
   ^
SyntaxError: invalid syntax
```

**Option Flags**

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in doctest directives, and may be passed to
the doctest command line interface via the `-o` option.

_New in version 3.4:_ The `-o` command line option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

**doctest.** `DONT_ACCEPT_TRUE_FOR_1`

By default, if an expected output block contains just 1, an actual output block containing just 1 or just `True` is considered to be a match, and similarly for 0 versus `False`. When `DONT_ACCEPT_TRUE_FOR_1` is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting “little integer” output still work in these cases. This option will probably go away, but not for several years.

**doctest.** `DONT_ACCEPT_BLANKLINE`

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When `DONT_ACCEPT_BLANKLINE` is specified, this substitution is not allowed.

**doctest.** `NORMALIZE_WHITESPACE`

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. `NORMALIZE_WHITESPACE` is especially useful when a line of
expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.ELLIPSIS

When specified, an ellipsis marker (...) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it’s best to keep usage of this simple. Complicated uses can lead to the same kinds of “oops, it matched too much!” surprises that .* is prone to in regular expressions.

doctest.IGNORE_EXCEPTION_DETAIL

When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting \nValueError: 42 \nwill pass if the actual exception raised is \nValueError: 3*14, but will fail, e.g., if TypeError is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```
>>> raise CustomError('message')
Traceback (most recent call last):
  CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
  my_module.CustomError: message
```

Note that ELLIPSIS can also be used to ignore the details of the exception message, but such a test may still fail based on whether
or not the module details are printed as part of the exception name. Using `IGNORE_EXCEPTION_DETAIL` and the details from Python 2.3 is also the only clear way to write a doctest that doesn’t care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support `doctest directives` and ignore them as irrelevant comments). For example:

```python
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say “does not” instead of “doesn’t”.

*Changed in version 3.2:* `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

**doctest.SKIP**

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example’s output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily “commenting out” examples.

**doctest.COMPARISON_FLAGS**

A bitmask or’ing together all the comparison flags above.
The second group of options controls how test failures are reported:

**doctest.REPORT_UDIFF**
When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

**doctest.REPORT_CDIFF**
When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

**doctest.REPORT_NDIFF**
When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

**doctest.REPORT_ONLY_FIRST_FAILURE**
When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When `REPORT_ONLY_FIRST_FAILURE` is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

**doctest.FAIL_FAST**
When specified, exit after the first failing example and don’t attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during
debugging, since examples after the first failure won’t even produce debugging output.

The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

*New in version 3.4.*

**doctest.** `REPORTING_FLAGS`

A bitmask or’ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn’t useful unless you intend to extend `doctest` internals via subclassing:

**doctest.** `register_optionflag(name)`

Create a new option flag with a given name, and return the new flag’s integer value. `register_optionflag()` can be used when subclassing `OutputChecker` or `DocTestRunner` to create new options that are supported by your subclasses. `register_optionflag()` should always be called using the following idiom:

```python
MY_FLAG = register_optionflag('MY_FLAG')
```

**Directives**

Doctest directives may be used to modify the option flags for an individual example. Doctest directives are special Python comments following an example's source code:

```python
directive ::= "#" "doctest:" directive
directive_options ::= directive_option ("," directive)
directive_option ::= on_or_off directive_option
```
on_or_off ::= "+" | "-
directive_option_name ::= "DONT_ACCEPT_BLANKLINE"

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example’s doctest directives modify doctest’s behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes:

```python
>>> print(list(range(20)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn’t have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```python
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```python
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

If multiple directive comments are used for a single example, then they are combined:
As the previous example shows, you can add ... lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```python
>>> print(list(range(20)))
...
[0, 1, ..., 18, 19]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via - in a directive can be useful.

**Warnings**

`doctest` is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```python
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do
Instead. Another is to do

```python
>>> d = sorted(foo().items())
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```python
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C() # the default repr() for instances embeds
<__main__.C instance at 0x00AC18F0>
```

The `ELLIPSIS` directive gives a nice approach for the last example:

```python
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```python
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7)  # safer
0.142857142857
>>> print(round(1./7, 6))  # much safer
```
Numbers of the form $I/2.**J$ are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.
Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections Simple Usage: Checking Examples in Docstrings and Simple Usage: Checking Examples in a Text File.

def doctest.testfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=DocTestParser(), encoding=None)

All arguments except `filename` are optional, and should be specified in keyword form.

Test examples in the file named `filename`. Return `(failure_count, test_count)`.

Optional argument `module_relative` specifies how the filename should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, `filename` should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then `filename` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.
Optional argument *name* gives the name of the test; by default, or if None, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is *False*.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if None, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like `dict.update()`: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if None, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if None, it's true if and only if '-v' is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).
Optional argument *optionflags* (default value 0) takes the bitwise OR of option flags. See section *Option Flags*.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., *DocTestParser()*).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

```
import doctest

doctest.testmod(m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)
```

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is *None*), starting with *m.__doc__*.

Also test examples reachable from dict *m.__test__*, if it exists and is not *None*. *m.__test__* maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return *(failure_count, test_count)*.
Optional argument *name* gives the name of the module; by default, or if None, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function `testfile()` above, except that *globs* defaults to `m.__dict__`.

```python
doctest.run_docstring_examples(f, globs, verbose=False, name="NoName", compileflags=None, optionflags=0)
```

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to "NoName".

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By
default, or if `None`, flags are deduced corresponding to the set of future features found in `globs`.

Optional argument `optionflags` works as for function `testfile()` above.
Unittest API

As your collection of doctest’ed modules grows, you'll want a way to run all their doctests systematically. `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a `load_tests()` function in your test module:

```python
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating `unittest.TestSuite` instances from text files and modules with doctests:

- `doctest.DocFileSuite(*paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None)`
  - Convert doctest tests from one or more text files to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.
Options may be provided as keyword arguments:

Optional argument `module_relative` specifies how the filenames in `paths` should be interpreted:

- If `module_relative` is `True` (the default), then each filename in `paths` specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If `module_relative` is `False`, then each filename in `paths` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in `paths`. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional argument `setUp` specifies a set-up function for the test suite. This is called before running the tests in each file. The `setUp` function will be passed a `DocTest` object. The `setUp` function can access the test globals as the `globs` attribute of the test passed.

Optional argument `tearDown` specifies a tear-down function for the test suite. This is called after running the tests in each file. The `tearDown` function will be passed a `DocTest` object. The `tearDown` function can access the test globals as the `globs` attribute of the test passed.
Optional argument `globs` is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, `globs` is a new empty dictionary.

Optional argument `optionflags` specifies the default doctest options for the tests, created by or-ing together individual option flags. See section Option Flags. See function `set_unitest_reportflags()` below for a better way to set reporting options.

Optional argument `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument `encoding` specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using `DocFileSuite()`.

doctest.`DocTestSuite`(module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, checker=None)

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument `module` provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not
specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above.

This function uses the same search technique as *testmod()*.

*Changed in version 3.5:* *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of doctest. DocTestCase instances, and DocTestCase is a subclass of *unittest.TestCase*. DocTestCase isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestTestCase`. This is important for a subtle reason: when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you're writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `unittest` to `doctest` test runners.

For this reason, `doctest` also supports a notion of `doctest` reporting flags specific to `unittest` support, via this function:

```python
doctest.set_unittest_reportflags(flags)
```

Set the `doctest` reporting flags to use.

Argument `flags` takes the bitwise OR of option flags. See section `Option Flags`. Only “reporting flags” can be used.

This is a module-global setting, and affects all future doctests run by module `unittest`: the `runTest()` method of `DocTestTestCase` looks at the option flags specified for the test case when the `DocTestTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`'s `unittest` reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `DocTestTestCase` instance was constructed, `doctest`'s `unittest` reporting flags are ignored.

The value of the `unittest` reporting flags in effect before the
function was called is returned by the function.
Advanced API

The basic API is a simple wrapper that’s intended to make doctest easy to use. It is fairly flexible, and should meet most users’ needs; however, if you require more fine-grained control over testing, or wish to extend doctest’s capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- **Example**: A single Python statement, paired with its expected output.
- **DocTest**: A collection of Examples, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- **DocTestFinder**: Finds all docstrings in a given module, and uses a DocTestParser to create a DocTest from every docstring that contains interactive examples.
- **DocTestParser**: Creates a DocTest object from a string (such as an object’s docstring).
- **DocTestRunner**: Executes the examples in a DocTest, and uses an OutputChecker to verify their output.
- **OutputChecker**: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:
DocTest Objects

class doctest.DocTest(examples, globs, name, filename, lineno, docstring)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples
A list of Example objects encoding the individual interactive Python examples that should be run by this test.

globs
The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in globs after the test is run.

name
A string name identifying the DocTest. Typically, this is the
name of the object or file that the test was extracted from.

**filename**

The name of the file that this `DocTest` was extracted from; or `None` if the filename is unknown, or if the `DocTest` was not extracted from a file.

**lineno**

The line number within `filename` where this `DocTest` begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

**docstring**

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

**Example Objects**

```python
class doctest.Example(source, want, exc_msg=None, lineno=0, indent=0, options=None)
```

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

`Example` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

**source**

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.
**want**

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). `want` ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

**exc_msg**

The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. `exc_msg` ends with a newline unless it's `None`. The constructor adds a newline if needed.

**lineno**

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

**indent**

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

**options**

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s `optionflags`). By default, no options are set.
DocTestFinder objects

class doctest.DocTestFinder(verbosede=False, 
parser=DocTestParser(), recurse=True, exclude_empty=True)

A processing class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. DocTests can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument verbose can be used to display the objects searched by the finder. It defaults to False (no output).

The optional argument parser specifies the DocTestParser object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument recurse is false, then DocTestFinder.find() will only examine the given object, and not any contained objects.

If the optional argument exclude_empty is false, then DocTestFinder.find() will include tests for objects with empty docstrings.

DocTestFinder defines the following method:

find(obj[, name][, module][, globs][, extraglobs])

Return a list of the DocTests that are defined by obj’s docstring, or by any of its contained objects’ docstrings.

The optional argument name specifies the object’s name; this
name will be used to construct names for the returned DocTests. If name is not specified, then \texttt{obj.__name__} is used.

The optional parameter \textit{module} is the module that contains the given object. If the module is not specified or is \texttt{None}, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if \textit{globs} is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than \textit{module} are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If \textit{module} is \texttt{False}, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if \textit{module} is \texttt{False}, or is \texttt{None} but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each \texttt{DocTest} is formed by combining \textit{globs} and \textit{extraglobs} (bindings in \textit{extraglobs} override bindings in \textit{globs}). A new shallow copy of the globals dictionary is created for each \texttt{DocTest}. If \textit{globs} is not specified, then it defaults to the module's \texttt{__dict__}, if specified, or \texttt{{}} otherwise. If \textit{extraglobs} is not specified, then it defaults to \texttt{{}}.

\textbf{DocTestParser objects}
class doctest.DocTestParser
A processing class used to extract interactive examples from a string, and use them to create a DocTest object.

DocTestParser defines the following methods:

**get_doctest**(string, globs, name, filename, lineno)
Examine all doctest examples from the given string, and collect them into a DocTest object.

globs, name, filename, and lineno are attributes for the new DocTest object. See the documentation for DocTest for more information.

**get_examples**(string, name='<string>')
Examine all doctest examples from the given string, and return them as a list of Example objects. Line numbers are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

**parse**(string, name='<string>')
Divide the given string into examples and intervening text, and return them as a list of alternating Examples and strings. Line numbers for the Examples are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

DocTestRunner objects

class doctest.DocTestRunner(checker=None, verbose=None, optionflags=0)
A processing class used to execute and verify the interactive
examples in a DocTest.

The comparison between expected outputs and actual outputs is done by an OutputChecker. This comparison may be customized with a number of option flags; see section Option Flags for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of OutputChecker to the constructor.

The test runner’s display output can be controlled in two ways. First, an output function can be passed to TestRunner.run(); this function will be called with strings that should be displayed. It defaults to sys.stdout.write. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods report_start(), report_success(), report_unexpected_exception(), and report_failure().

The optional keyword argument checker specifies the OutputChecker object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument verbose controls the DocTestRunner’s verbosity. If verbose is True, then information is printed about each example, as it is run. If verbose is False, then only failures are printed. If verbose is unspecified, or None, then verbose output is used iff the command-line switch -v is used.

The optional keyword argument optionflags can be used to control
how the test runner compares expected output to actual output, and how it displays failures. For more information, see section Option Flags.

**DocTestParser** defines the following methods:

**report_start**(out, test, example)
Report that the test runner is about to process the given example. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

*example* is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to DocTestRunner.run().

**report_success**(out, test, example, got)
Report that the given example ran successfully. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

*example* is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to DocTestRunner.run().

**report_failure**(out, test, example, got)
Report that the given example failed. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

*example* is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. 
out is the output function that was passed to DocTestRunner.run().

report_unexpected_exception(out, test, example, exc_info)
Report that the given example raised an unexpected exception. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

docs example is the example about to be processed. exc_info is a tuple containing information about the unexpected exception (as returned by sys.exc_info()). test is the test containing example. out is the output function that was passed to DocTestRunner.run().

run(test, compileflags=None, out=None, clear_globs=True)
Run the examples in test (a DocTest object), and display the results using the writer function out.

The examples are run in the namespace test.globs. If clear_globs is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use clear_globs=False.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to globs.

The output of each example is checked using the DocTestRunner’s output checker, and the results are
formatted by the DocTestRunner.report_*( ) methods.

**summarize**(verbose=None)
Print a summary of all the test cases that have been run by this
DocTestRunner, and return a named tuple
TestResults(failed, attempted).

The optional verbose argument controls how detailed the
summary is. If the verbosity is not specified, then the
DocTestRunner's verbosity is used.

**OutputChecker objects**

class doctest. **OutputChecker**
A class used to check the whether the actual output from a doctest
example matches the expected output. **OutputChecker** defines
two methods: check_output( ), which compares a given pair of
outputs, and returns true if they match; and
output_difference( ), which returns a string describing the
differences between two outputs.

**OutputChecker** defines the following methods:

**check_output**(want , got , optionflags)
Return True iff the actual output from an example (got)
matches the expected output (want). These strings are always
considered to match if they are identical; but depending on what
option flags the test runner is using, several non-exact match
types are also possible. See section **Option Flags** for more
information about option flags.

**output_difference**(example , got , optionflags)
Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*. 
Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, `pdb`.

- The `DebugRunner` class is a subclass of `DocTestRunner` that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.

- The `unittest` cases generated by `DocTestSuite()` support the `debug()` method defined by `unittest.TestCase`.

- You can add a call to `pdb.set_trace()` in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
""
>>> def f(x):
...    g(x*2)
>>> def g(x):
...    print(x+3)
...    import pdb; pdb.set_trace()
>>> f(3)
g
""
```

Then an interactive Python session may look like this:
Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1    def g(x):
  2        print(x+3)
  3   ->   import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1    def f(x):
  2    ->   g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>> 
```

doctest.script_from_examples(s)

   Convert text with examples to a script.
Argument $s$ is a string containing doctest examples. The string is converted to a Python script, where doctest examples in $s$ are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```python
import doctest
print(doctest.script_from_examples(r"
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"
))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

**doctest.testsource**(module, name)

Convert the doctest for an object to a script.

Argument `module` is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument
name is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for script_from_examples() above. For example, if module a.py contains a top-level function f(), then

```python
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function f()’s docstring, with doctests converted to code, and the rest placed in comments.

**doctest.debug(module, name, pm=False)**

Debug the doctests for an object.

The module and name arguments are the same as for function testsource() above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, pdb.

A shallow copy of module.__dict__ is used for both local and global execution context.

Optional argument pm controls whether post-mortem debugging is used. If pm has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via pdb.post_mortem(), passing the traceback object from the unhandled exception. If pm is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate exec() call to pdb.run().

**doctest.debug_src(src, pm=False, globs=None)**
Debug the doctests in a string.

This is like function `debug()` above, except that a string containing doctest examples is specified directly, via the `src` argument.

Optional argument `pm` has the same meaning as in function `debug()` above.

Optional argument `globals` gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially `DebugRunner`'s docstring (which is a doctest!) for more details:

```python
class doctest.DebugRunner(checker=None, verbose=None, optionflags=0)
```

A subclass of `DocTestRunner` that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an `UnexpectedException` exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a `DocTestFailure` exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for `DocTestRunner` in section Advanced API.

There are two exceptions that may be raised by `DebugRunner`
instances:

exception doctest.**DocTestFailure**(test, example, got)

An exception raised by **DocTestRunner** to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

**DocTestFailure** defines the following attributes:

**DocTestFailure.test**

The **DocTest** object that was being run when the example failed.

**DocTestFailure.example**

The **Example** that failed.

**DocTestFailure.got**

The example's actual output.

exception doctest.**UnexpectedException**(test, example, exc_info)

An exception raised by **DocTestRunner** to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

**UnexpectedException** defines the following attributes:

**UnexpectedException.test**

The **DocTest** object that was being run when the example failed.

**UnexpectedException.example**

The **Example** that failed.
**UnexpectedException**. **exc_info**

A tuple containing information about the unexpected exception, as returned by `sys.exc_info()`.
Soapbox

As mentioned in the introduction, doctest has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my doctest examples stops working after a “harmless” change.

Doctest also makes an excellent tool for regression testing, especially if you don’t skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what’s actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It’s true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper
than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It’s a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```python
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            
```
obj = globals()[name]

else:
    obj = __test__[name]
doctest.run_docstring_examples(obj, globals(),
    optionflags=flags)

else:
    fail, total = doctest.testmod(optionflags=flags)
print("{} failures out of {} tests".format(fail, total))

Footnotes

Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.
unittest — Unit testing framework

Source code: Lib/unittest/__init__.py

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite

A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.
A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

See also:

**Module doctest**
Another test-support module with a very different flavor.

**Simple Smalltalk Testing: With Patterns**
Kent Beck’s original paper on testing frameworks using the pattern shared by unittest.

**Nose and pytest**
Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

**The Python Testing Tools Taxonomy**
An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

**Testing in Python Mailing List**
A special-interest-group for discussion of testing, and testing tools, in Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins or Hudson.
Basic example

The `unittest` module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```python
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing `unittest.TestCase`. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.
The crux of each test is a call to `assertEqual()` to check for an expected result; `assertTrue()` or `assertFalse()` to verify a condition; or `assertRaises()` to verify that a specific exception gets raised. These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.

The `setUp()` and `tearDown()` methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```plaintext
Ran 3 tests in 0.000s
OK
```

Passing the `-v` option to your test script will instruct `unittest.main()` to enable a higher level of verbosity, and produce the following output:

```plaintext
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

Ran 3 tests in 0.001s
OK
```
The above examples show the most commonly used `unittest` features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.
Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the ‘.py’ and converting path separators into ‘.’. If you want to execute a test file that isn’t importable as a module you should execute the file directly instead.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

When executed without arguments Test Discovery is started:

```
python -m unittest
```
For a list of all the command-line options:

```
python -m unittest -h
```

*Changed in version 3.2:* In earlier versions it was only possible to run individual test methods and not modules or classes.

## Command-line options

`unittest` supports these command-line options:

- **-b, --buffer**
  The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

- **-c, --catch**
  Control-C during the test run waits for the current test to end and then reports all the results so far. A second Control-C raises the normal `KeyboardInterrupt` exception.

  See [Signal Handling](#) for the functions that provide this functionality.

- **-f, --failfast**
  Stop the test run on the first error or failure.

- **-k**
  Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match of the given patterns are included.

  Patterns that contain a wildcard character (*) are matched against
the test name using `fnmatch.fnmatchcase()`; otherwise simple case-sensitive substring matching is used.

Patterns are matched against the fully qualified test method name as imported by the test loader.

For example, `-k foo` matches `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`.

**--locals**

Show local variables in tracebacks.

*New in version 3.2:* The command-line options `-b`, `-c` and `-f` were added.

*New in version 3.5:* The command-line option `--locals`.

*New in version 3.7:* The command-line option `-k`.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.
Test Discovery

New in version 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages (including namespace packages) importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in TestLoader.discover(), but can also be used from the command line. The basic command-line usage is:

```bash
cd project_directory
python -m unittest discover
```

**Note:** As a shortcut, python -m unittest is the equivalent of python -m unittest discover. If you want to pass arguments to test discovery the discover sub-command must be used explicitly.

The discover sub-command has the following options:

- **-v, --verbose**
  
  Verbose output

- **-s, --start-directory directory**
  
  Directory to start discovery (default)

- **-p, --pattern pattern**
  
  Pattern to match test files (test*.py default)
-t, --top-level-directory directory
  Top level directory of project (defaults to start directory)

The -s, -p, and -t options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```bash
python -m unittest discover -s project_directory -p
python -m unittest discover project_directory "*_test.py"
```

As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

**Caution:** Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the `load_tests protocol`.

*Changed in version 3.4:* Test discovery supports `namespace`
packages.
Organizing test code

The basic building blocks of unit testing are test cases — single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```python
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the `assert*()` methods provided by the `TestCase` base class. If the test fails, an exception will be raised with an explanatory message, and `unittest` will identify the test case as a `failure`. Any other exceptions will be treated as `errors`.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test.
we run:

```python
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50, 50), 'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100, 150)
        self.assertEqual(self.widget.size(), (100, 150), 'wrong size after resize')

Note: The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```
```python
def tearDown(self):
    self.widget.dispose()
```

If `setUp()` succeeded, `tearDown()` will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a test fixture. A new TestCase instance is created as a unique test fixture used to execute each individual test method. Thus `setUp()`, `tearDown()`, and `__init__()` will be called once per test.

It is recommended that you use TestCase implementations to group tests together according to the features they test. `unittest` provides a mechanism for this: the test suite, represented by `unittest`'s `TestSuite` class. In most cases, calling `unittest.main()` will do the right thing and collect all the module's test cases for you and execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```python
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite
```

```python
if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there
are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.
Re-using old test code

Some users will find that they have existing test code that they would like to run from `unittest`, without converting every old test function to a `TestCase` subclass.

For this reason, `unittest` provides a `FunctionTest` class. This subclass of `TestCase` can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```python
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```python
testcase = unittest.FunctionTest(testSomething,
                                 setUp=makeSomethingDB,
                                 tearDown=deleteSomethingDB)
```

**Note:** Even though `FunctionTest` can be used to quickly convert an existing test base over to a `unittest`-based system, this approach is not recommended. Taking the time to set up proper `TestCase` subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the
doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests.
Skipping tests and expected failures

*New in version 3.1.*

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an “expected failure,” a test that is broken and will fail, but shouldn’t be counted as a failure on a `TestResult`.

Skipping a test is simply a matter of using the `skip()` decorator or one of its conditional variants.

Basic skipping looks like this:

```python
class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library")
    def test_format(self):
        # Tests that work for only a certain version
        pass

    @unittest.skipUnless(sys.platform.startswith("win"),
                         "windows specific testing code")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library'
```
Classes can be skipped just like methods:

```python
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

`TestCase.setUp()` can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the `expectedFailure()` decorator.

```python
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls `skip()` on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```python
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```
The following decorators implement test skipping and expected failures:

@unittest.skip(reason)
   Unconditionally skip the decorated test. reason should describe why the test is being skipped.

@unittest.skipIf(condition, reason)
   Skip the decorated test if condition is true.

@unittest.skipUnless(condition, reason)
   Skip the decorated test unless condition is true.

@unittest.expectedFailure
   Mark the test as an expected failure. If the test fails it will be considered a success. If the test passes, it will be considered a failure.

exception unittest.SkipTest(reason)
   This exception is raised to skip a test.

   Usually you can use TestCase.skipTest() or one of the skipping decorators instead of raising this directly.

Skipped tests will not have setUp() or tearDown() run around them. Skipped classes will not have setUpClass() or tearDownClass() run. Skipped modules will not have setUpModule() or tearDownModule() run.
Distinguishing test iterations using subtests

New in version 3.4.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the \texttt{subTest()} context manager.

For example, the following test:

```python
class NumbersTest(unittest.TestCase):
    def test_even(self):
        
        Test that numbers between 0 and 5 are all even
        
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will produce the following output:

```
FAIL: test_even (__main__.NumbersTest) (i=1)

Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

FAIL: test_even (__main__.NumbersTest) (i=3)
```
Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of i wouldn’t be displayed:
Classes and functions

This section describes in depth the API of `unittest`.

Test cases

```
class unittest.TestCase(methodName='runTest')
```

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named `methodName`. In most uses of `TestCase`, you will neither change the `methodName` nor reimplement the default `runTest()` method.

*Changed in version 3.2:* `TestCase` can be instantiated successfully without providing a `methodName`. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

```
s setUp()
```
Method called to prepare the test fixture. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

**tearDown()**

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

**setUpClass()**

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

```python
@classmethod
def setUpClass(cls):
    ...
```

See [Class and Module Fixtures](#) for more details.

*New in version 3.2.*

**tearDownClass()**

A class method called after tests in an individual class have run.
tearDownClass is called with the class as the only argument and must be decorated as a `classmethod()`:

```python
@classmethod
def tearDownClass(cls):
    ...
```

See [Class and Module Fixtures](#) for more details.

*New in version 3.2.*

**run**(result=None)

Run the test, collecting the result into the `TestResult` object passed as `result`. If `result` is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`’s caller.

The same effect may be had by simply calling the `TestCase` instance.

*Changed in version 3.3:* Previous versions of `run` did not return the result. Neither did calling an instance.

**skipTest**(reason)

Calling this during a test method or `setUp()` skips the current test. See [Skipping tests and expected failures](#) for more information.

*New in version 3.1.*

**subTest**(msg=None, **params)

Return a context manager which executes the enclosed code
block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See [Distinguishing test iterations using subtests](#) for more information.

*New in version 3.4.*

**debug()**

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The **TestCase** class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEqual(a, b)</td>
<td>a == b</td>
<td></td>
</tr>
<tr>
<td>assertNotEqual(a, b)</td>
<td>a != b</td>
<td></td>
</tr>
<tr>
<td>assertTrue(x)</td>
<td>bool(x) is True</td>
<td></td>
</tr>
<tr>
<td>assertFalse(x)</td>
<td>bool(x) is False</td>
<td></td>
</tr>
<tr>
<td>assertIs(a, b)</td>
<td>a is b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNot(a, b)</td>
<td>a is not b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNone(x)</td>
<td>x is None</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNotNone(x)</td>
<td>x is not None</td>
<td>3.1</td>
</tr>
<tr>
<td>Method</td>
<td>Expression</td>
<td>Version</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>assertIn(a, b)</code></td>
<td><code>a in b</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertNotIn(a, b)</code></td>
<td><code>a not in b</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertIsInstance(a, b)</code></td>
<td><code>isinstance(a, b)</code></td>
<td>3.2</td>
</tr>
<tr>
<td><code>assertNotIsInstance(a, b)</code></td>
<td><code>not isinstance(a, b)</code></td>
<td>3.2</td>
</tr>
</tbody>
</table>

All the assert methods accept a `msg` argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the `msg` keyword argument can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

**`assertEqual(first, second, msg=None)`**

Test that `first` and `second` are equal. If the values do not compare equal, the test will fail.

In addition, if `first` and `second` are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

*Changed in version 3.1:* Added the automatic calling of type-specific equality function.

*Changed in version 3.2:* `assertMultiLineEqual()` added as the default type equality function for comparing strings.
assertNotEqual(first, second, msg=None)
Test that first and second are not equal. If the values do compare equal, the test will fail.

assertTrue(expr, msg=None)
assertFalse(expr, msg=None)
Test that expr is true (or false).

Note that this is equivalent to bool(expr) is True and not to expr is True (use assertIs(expr, True) for the latter). This method should also be avoided when more specific methods are available (e.g. assertEqual(a, b) instead of assertTrue(a == b)), because they provide a better error message in case of failure.

assertIs(first, second, msg=None)
assertIsNot(first, second, msg=None)
Test that first and second evaluate (or don’t evaluate) to the same object.

New in version 3.1.

assertIsNone(expr, msg=None)
assertIsNotNone(expr, msg=None)
Test that expr is (or is not) None.

New in version 3.1.

assertIn(first, second, msg=None)
assertNotIn(first, second, msg=None)
Test that first is (or is not) in second.

New in version 3.1.
**assertIsInstance**(obj, cls, msg=None)
**assertNotIsInstance**(obj, cls, msg=None)

Test that obj is (or is not) an instance of cls (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls).

New in version 3.2.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertRaises(exc, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>exc</code></td>
<td></td>
</tr>
<tr>
<td><code>assertRaisesRegex(exc, r, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>exc</code> and the message matches regex <code>r</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertWarns(warn, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>warn</code></td>
<td>3.2</td>
</tr>
<tr>
<td><code>assertWarnsRegex(warn, r, fun, *args, **kwds)</code></td>
<td><code>fun(*args, **kwds)</code> raises <code>warn</code> and the message matches regex <code>r</code></td>
<td>3.2</td>
</tr>
<tr>
<td><code>assertLogs(logger, level)</code></td>
<td>The <strong>with</strong> block logs on <code>logger</code> with minimum <code>level</code></td>
<td>3.4</td>
</tr>
</tbody>
</table>

**assertRaises**(exception, callable, *args, **kwds)
**assertRaises**(exception, *, msg=None)
Test that an exception is raised when `callable` is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if `exception` is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as `exception`.

If only the `exception` and possibly the `msg` arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```python
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, `assertRaises()` accepts the additional keyword argument `msg`.

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```python
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

*Changed in version 3.1:* Added the ability to use `assertRaises()` as a context manager.

*Changed in version 3.2:* Added the exception attribute.

*Changed in version 3.3:* Added the `msg` keyword argument when used as a context manager.
assertRaisesRegex(exception, regex, callable, *args, **kwds)

assertRaisesRegex(exception, regex, *, msg=None)

Like assertRaises() but also tests that regex matches on the string representation of the raised exception. regex may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```python
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'", int, 'XYZ')
```

or:

```python
with self.assertRaisesRegex(ValueError, 'literal
int('XYZ')
```

New in version 3.1: under the name `assertRaisesRegexp`.

Changed in version 3.2: Renamed to `assertRaisesRegex()`.

Changed in version 3.3: Added the msg keyword argument when used as a context manager.

assert_warns(warning, callable, *args, **kwds)

assertWarns(warning, *, msg=None)

Test that a warning is triggered when `callable` is called with any positional or keyword arguments that are also passed to `assertWarns()`. The test passes if `warning` is triggered and fails if it isn’t. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as `warnings`.

```python
self.assertWarnsRegexp(ValueError, "invalid literal for.*XYZ'", int, 'XYZ')
```
If only the `warning` and possibly the `msg` arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```python
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, `assertWarnings()` accepts the additional keyword argument `msg`.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```python
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

*New in version 3.2.*

*Changed in version 3.3:* Added the `msg` keyword argument when used as a context manager.

```python
assertWarnsRegex(warning, regex, callable, *args, **kwds)
assertWarnsRegex(warning, regex, *, msg=None)
```

Like `assertWarns()` but also tests that `regex` matches on the message of the triggered warning. `regex` may be a regular
expression object or a string containing a regular expression suitable for use by `re.search()`. Example:

```python
def self.assertWarnsRegex(cls, expected_warning_class, expected_regex, *args, **kwargs):

```

or:

```python
with self.assertWarnsRegex(cls, expected_warning_class, expected_regex, *args, **kwargs):

```

New in version 3.2.

Changed in version 3.3: Added the `msg` keyword argument when used as a context manager.

```
self.assertLogs(self.logger, level=self.INFO)

```

A context manager to test that at least one message is logged on the `logger` or one of its children, with at least the given `level`.

If given, `logger` should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages.

If given, `level` should be either a numeric logging level or its string equivalent (for example either "ERROR" or `logging.ERROR`). The default is `logging.INFO`.

The test passes if at least one message emitted inside the `with` block matches the `logger` and `level` conditions, otherwise it fails.
The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

**records**
A list of `logging.LogRecord` objects of the matching log messages.

**output**
A list of `str` objects with the formatted output of matching messages.

Example:

```python
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message', 'ERROR:foo.bar:second message'])
```

*New in version 3.4.*

There are also other methods used to perform more specific checks, such as:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertAlmostEqual(a, b)</code></td>
<td><code>round(a-b, 7) == 0</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual(a, b)</code></td>
<td><code>round(a-b, 7) != 0</code></td>
<td></td>
</tr>
<tr>
<td><code>assertGreater(a, b)</code></td>
<td><code>a &gt; b</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertGreaterEqual(a, b)</code></td>
<td><code>a &gt;= b</code></td>
<td>3.1</td>
</tr>
</tbody>
</table>
### b)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertLess(a, b)</code></td>
<td>$a &lt; b$</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertLessEqual(a, b)</code></td>
<td>$a \leq b$</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertRegex(s, r)</code></td>
<td><code>r.search(s)</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertNotRegex(s, r)</code></td>
<td><code>not r.search(s)</code></td>
<td>3.2</td>
</tr>
<tr>
<td><code>assertCountEqual(a, b)</code></td>
<td>$a$ and $b$ have the same elements in the same number, regardless of their order</td>
<td>3.2</td>
</tr>
</tbody>
</table>

#### assertAlmostEqual

Test that `first` and `second` are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal places (default 7), and comparing to zero. Note that these methods round the values to the given number of decimal places (i.e. like the `round()` function) and not significant digits.

If `delta` is supplied instead of `places` then the difference between `first` and `second` must be less or equal to (or greater than) `delta`.

Supplying both `delta` and `places` raises a `TypeError`.

*Changed in version 3.2:* `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the `delta` keyword argument.
assertGreater(first, second, msg=None)
assertGreaterEqual(first, second, msg=None)
assertLess(first, second, msg=None)
assertLessEqual(first, second, msg=None)

Test that first is respectively >, >=, < or <= than second depending on the method name. If not, the test will fail:

```python
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

New in version 3.1.

assertRegex(text, regex, msg=None)
assertNotRegex(text, regex, msg=None)

Test that a regex search matches (or does not match) text. In case of failure, the error message will include the pattern and the text (or the pattern and the part of text that unexpectedly matched). regex may be a regular expression object or a string containing a regular expression suitable for use by re.search().

New in version 3.1: under the name assertRegexpMatches.

Changed in version 3.2: The method assertRegexpMatches() has been renamed to assertRegex().

New in version 3.2: assertNotRegex().

New in version 3.5: The name assertNotRegexpMatches is a deprecated alias for assertNotRegex().

assertCountEqual(first, second, msg=None)
Test that sequence `first` contains the same elements as `second`, regardless of their order. When they don’t, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing `first` and `second`. It verifies whether each element has the same count in both sequences. Equivalent to:

```
assertEqual(Counter(list(first)), Counter(list(second)))
```

but works with sequences of unhashable objects as well.

_New in version 3.2._

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it’s also possible to register new methods using `addTypeEqualityFunc()`:

```
addTypeEqualityFunc(typeobj, function)
```

Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same `typeobj` (not subclasses) compare equal. `function` must take two positional arguments and a third `msg=None` keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

_New in version 3.1._
The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

<table>
<thead>
<tr>
<th>Method</th>
<th>Used to compare</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertMultiLineEqual(a, b)</code></td>
<td>strings</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertSequenceEqual(a, b)</code></td>
<td>sequences</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertListEqual(a, b)</code></td>
<td>lists</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertTupleEqual(a, b)</code></td>
<td>tuples</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertSetEqual(a, b)</code></td>
<td>sets or frozensets</td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertDictEqual(a, b)</code></td>
<td>dicts</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**assertMultiLineEqual** *(first, second, msg=None)*

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

*New in version 3.1.*

**assertSequenceEqual** *(first, second, msg=None, seq_type=None)*

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's
used to implement `assertListEqual()` and `assertTupleEqual()`.

New in version 3.1.

`assertListEqual(first, second, msg=None)`

`assertTupleEqual(first, second, msg=None)`

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

New in version 3.1.

`assertSetEqual(first, second, msg=None)`

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of `first` or `second` does not have a `set.difference()` method.

New in version 3.1.

`assertDictEqual(first, second, msg=None)`

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

New in version 3.1.
Finally the `TestCase` provides the following methods and attributes:

**fail** *(msg=)*None* *

Signals a test failure unconditionally, with *msg* or None for the error message.

**failureException**

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to “play fair” with the framework. The initial value of this attribute is `AssertionError`.

**longMessage**

This class attribute determines what happens when a custom failure message is passed as the *msg* argument to an `assertXYY` call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, `self.longMessage`, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

*New in version 3.1.*

**maxDiff**

This attribute controls the maximum length of diffs output by
assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are
assertSequenceEqual() (including all the sequence comparison methods that delegate to it),
assertDictEqual() and assertMultiLineEqual().

Setting maxDiff to None means that there is no maximum length of diffs.

*New in version 3.2.*

Testing frameworks can use the following methods to collect information on the test:

**countTestCases()**

Return the number of tests represented by this test object. For TestCase instances, this will always be 1.

**defaultTestResult()**

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the run() method).

For TestCase instances, this will always be an instance of TestResult; subclasses of TestCase should override this as necessary.

**id()**

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

**shortDescription()**
Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or `None`.

*Changed in version 3.1:* In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the `TextTestResult` in Python 3.2.

`addCleanup(function, *args, **kwargs)`

Add a function to be called after `tearDown()` to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into `addCleanup()` when they are added.

If `setUp()` fails, meaning that `tearDown()` is not called, then any cleanup functions added will still be called.

*New in version 3.1.*

`doCleanups()`

This method is called unconditionally after `tearDown()`, or after `setUp()` if `setUp()` raises an exception.

It is responsible for calling all the cleanup functions added by `addCleanup()`. If you need cleanup functions to be called *prior* to `tearDown()` then you can call `doCleanups()` yourself.

`doCleanups()` pops methods off the stack of cleanup
functions one at a time, so it can be called at any time.

*New in version 3.1.*

class unittest.FunctionTestCase(testFunc, setUp=None, tearDown=None, description=None)

This class implements the portion of the `TestCase` interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a `unittest`-based test framework.

**Deprecated aliases**

For historical reasons, some of the `TestCase` methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Deprecated alias</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertEqual()</code></td>
<td><code>failUnlessEqual</code></td>
<td><code>assertEq</code></td>
</tr>
<tr>
<td><code>assertNotEqual()</code></td>
<td><code>failIfEqual</code></td>
<td><code>assertNotEq</code></td>
</tr>
<tr>
<td><code>assertTrue()</code></td>
<td><code>failUnless</code></td>
<td><code>assert_</code></td>
</tr>
<tr>
<td><code>assertFalse()</code></td>
<td><code>failIf</code></td>
<td></td>
</tr>
<tr>
<td><code>assertRaises()</code></td>
<td><code>failUnlessRaises</code></td>
<td></td>
</tr>
<tr>
<td><code>assertAlmostEqual()</code></td>
<td><code>failUnlessAlmostEqual</code></td>
<td><code>assertAlm</code></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual()</code></td>
<td><code>failIfAlmostEqual</code></td>
<td><code>assertNotAlm</code></td>
</tr>
<tr>
<td><code>assertRegex()</code></td>
<td></td>
<td><code>assertReg</code></td>
</tr>
<tr>
<td><code>assertNotRegex()</code></td>
<td></td>
<td><code>assertNotReg</code></td>
</tr>
<tr>
<td><code>assertRaisesRegex()</code></td>
<td></td>
<td><code>assertRaise</code></td>
</tr>
</tbody>
</table>
*Deprecated since version 3.1:* the fail* aliases listed in the second column.

*Deprecated since version 3.2:* the assert* aliases listed in the third column.

*Deprecated since version 3.2:* `assertRegexpMatches` and `assertRaisesRegexp` have been renamed to `assertRegex()` and `assertRaisesRegex()`.

*Deprecated since version 3.5:* the `assertNotRegexpMatches` name in favor of `assertNotRegex()`.

**Grouping tests**

```python
class unittest.TestSuite(tests=())
```

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.

If `tests` is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to `TestSuite` instances:
addTest(test)
Add a TestCase or TestSuite to the suite.

addTests(tests)
Add all the tests from an iterable of TestCase and TestSuite instances to this test suite.

This is equivalent to iterating over tests, calling addTest() for each element.

TestSuite shares the following methods with TestCase:

run(result)
Run the tests associated with this suite, collecting the result into the test result object passed as result. Note that unlike testCase.run(), TestSuite.run() requires the result object to be passed in.

debug()
Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()
Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()
Tests grouped by a TestSuite are always accessed by iteration. Subclasses can lazily provide tests by overriding __iter__(). Note that this method may be called several
times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before TestSuite.run() must be the same for each call iteration. After TestSuite.run(), callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides TestSuite._removeTestAtIndex() to preserve test references.

*Changed in version 3.2:* In earlier versions the TestSuite accessed tests directly rather than through iteration, so overriding __iter__() wasn’t sufficient for providing tests.

*Changed in version 3.4:* In earlier versions the TestSuite held references to each TestCase after TestSuite.run(). Subclasses can restore that behavior by overriding TestSuite._removeTestAtIndex().

In the typical usage of a TestSuite object, the run() method is invoked by a TestRunner rather than by the end-user test harness.

### Loading and running tests

**class unittest.TestLoader**

The TestLoader class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the unittest module provides an instance that can be shared as unittest.defaultTestLoader. Using a subclass or instance, however, allows customization of some configurable properties.
TestLoader objects have the following attributes:

**errors**
A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant a method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

*New in version 3.5.*

TestLoader objects have the following methods:

**loadTestsFromTestCase**(testCaseClass)
Return a suite of all test cases contained in the TestCase-derived testCaseClass.

A test case instance is created for each method named by getTestCaseNames(). By default these are the method names beginning with test. If getTestCaseNames() returns no methods, but the runTest() method is implemented, a single test case is created for that method instead.

**loadTestsFromModule**(module, pattern=None)
Return a suite of all test cases contained in the given module. This method searches module for classes derived from TestCase and creates an instance of the class for each test method defined for the class.

**Note:** While using a hierarchy of TestCase-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to
be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the `load_tests protocol`. The pattern argument is passed as the third argument to `load_tests`.

*Changed in version 3.2:* Support for `load_tests` added.

*Changed in version 3.5:* The undocumented and unofficial `use_load_tests` default argument is deprecated and ignored, although it is still accepted for backward compatibility. The method also now accepts a keyword-only argument `pattern` which is passed to `load_tests` as the third argument.

`loadTestsFromName(name, module=None)`

Return a suite of all test cases given a string specifier.

The specifier `name` is a “dotted name” that may resolve either to a module, a test case class, a test method within a test case class, a `TestSuite` instance, or a callable object which returns a `TestCase` or `TestSuite` instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as “a test method within a test case class”, rather than “a callable object”.

For example, if you have a module `SampleTests` containing a `TestCase`-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test
Using the specifier 'SampleTests.SampleTestCase.test_two' would cause it to return a test suite which will run only the test_two() test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves name relative to the given module.

*Changed in version 3.5:* If an `ImportError` or `AttributeError` occurs while traversing name then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by `self.errors`.

**loadTestsFromNames** *(names, module=None)*
Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

**getTestCaseNames** *(testCaseClass)*
Return a sorted sequence of method names found within `testCaseClass`; this should be a subclass of `TestCase`.

**discover** *(start_dir, pattern='test*.py', top_level_dir=None)*
Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match `pattern` will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.
All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the `load_tests` function itself calls `loader.discover`.

If `load_tests` exists then discovery does not recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. `top_level_dir` is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

`start_dir` can be a dotted module name as well as a directory.

*New in version 3.2.*

*Changed in version 3.4:* Modules that raise `SkipTest` on import are recorded as skips, not errors. Discovery works for namespace packages. Paths are sorted before being imported
so that execution order is the same even if the underlying file system’s ordering is not dependent on file name.

*Changed in version 3.5:* Found packages are now checked for `load_tests` regardless of whether their path matches `pattern`, because it is impossible for a package name to match the default pattern.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

**testMethodPrefix**
String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

**sortTestMethodsUsing**
Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

**suiteClass**
Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the `TestSuite` class.

This affects all the `loadTestsFrom*()` methods.

**testNamePatterns**
List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-v`
If this attribute is not None (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using `fnmatch.fnmatchcase()`, so unlike patterns passed to the `-v` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*()` methods.

*New in version 3.7.*

```python
class unittest.TestResult
    This class is used to compile information about which tests have succeeded and which have failed.

    A TestResult object stores the results of a set of tests. The TestCase and TestSuite classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

    Testing frameworks built on top of unittest may want access to the TestResult object generated by running a set of tests for reporting purposes; a TestResult instance is returned by the TestRunner.run() method for this purpose.

    TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

    errors
        A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test
which raised an unexpected exception.

**failures**
A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

**skipped**
A list containing 2-tuples of `TestCase` instances and strings holding the reason for skipping the test.

*New in version 3.1.*

**expectedFailures**
A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

**unexpectedSuccesses**
A list containing `TestCase` instances that were marked as expected failures, but succeeded.

**shouldStop**
Set to `True` when the execution of tests should stop by `stop()`.

**testsRun**
The total number of tests run so far.

**buffer**
If set to true, `sys.stdout` and `sys.stderr` will be buffered in between `startTest()` and `stopTest()` being called.
Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

*New in version 3.2.*

**failfast**

If set to true `stop()` will be called on the first failure or error, halting the test run.

*New in version 3.2.*

**tb_locals**

If set to true then local variables will be shown in tracebacks.

*New in version 3.5.*

**wasSuccessful()**

Return `True` if all tests run so far have passed, otherwise returns `False`.

*Changed in version 3.4*: Returns `False` if there were any `unexpectedSuccesses` from tests marked with the `expectedFailure()` decorator.

**stop()**

This method can be called to signal that the set of tests being run should be aborted by setting the `shouldStop` attribute to `True`. TestRunner objects should respect this flag and return without running any additional tests.

For example, this feature is used by the `TextTestRunner` class to stop the test framework when the user signals an
interrupt from the keyboard. Interactive tools which provide TestRunner implementations can use this in a similar manner.

The following methods of the TestResult class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

**startTest**(test)

Called when the test case test is about to be run.

**stopTest**(test)

Called after the test case test has been executed, regardless of the outcome.

**startTestRun**()

Called once before any tests are executed.

*New in version 3.1.*

**stopTestRun**()

Called once after all tests are executed.

*New in version 3.1.*

**addError**(test, err)

Called when the test case test raises an unexpected exception. err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends a tuple (test, formatted_err) to the instance's errors attribute, where
formatted_err is a formatted traceback derived from err.

addFailure(test, err)
Called when the test case test signals a failure. err is a tuple of the form returned by sys.exc_info(): (type, value, traceback).

The default implementation appends a tuple (test, formatted_err) to the instance's failures attribute, where formatted_err is a formatted traceback derived from err.

addSuccess(test)
Called when the test case test succeeds.

The default implementation does nothing.

addSkip(test, reason)
Called when the test case test is skipped. reason is the reason the test gave for skipping.

The default implementation appends a tuple (test, reason) to the instance's skipped attribute.

addExpectedFailure(test, err)
Called when the test case test fails, but was marked with the expectedFailure() decorator.

The default implementation appends a tuple (test, formatted_err) to the instance's expectedFailures attribute, where formatted_err is a formatted traceback derived from err.

addUnexpectedSuccess(test)
Called when the test case `test` was marked with the `expectedFailure()` decorator, but succeeded.

The default implementation appends the test to the instance's `unexpectedSuccesses` attribute.

### addSubTest(test, subtest, outcome)
Called when a subtest finishes. `test` is the test case corresponding to the test method. `subtest` is a custom `TestCase` instance describing the subtest.

If `outcome` is `None`, the subtest succeeded. Otherwise, it failed with an exception where `outcome` is a tuple of the form returned by `sys.exc_info()`:

```
(type, value, traceback)
```

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

*New in version 3.4.*

class unittest.TextTestResult(stream, descriptions, verbosity)
A concrete implementation of `TestResult` used by the `TextTestRunner`.

*New in version 3.2:* This class was previously named `_TextTestResult`. The old name still exists as an alias but is deprecated.

`unittest.defaultTestLoader`
Instance of the `TestLoader` class intended to be shared. If no customization of the `TestLoader` is needed, this instance can be used instead of repeatedly creating new instances.
class unittest.TextTestRunner(stream=None, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None, warnings=None, *, tb_locals=False)

A basic test runner implementation that outputs results to a stream. If `stream` is `None`, the default, `sys.stderr` is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept **kwargs as the interface to construct runners changes when features are added to unittest.

By default this runner shows DeprecationWarning, PendingDeprecationWarning, ResourceWarning and ImportWarning even if they are ignored by default. Deprecation warnings caused by deprecated unittest methods are also special-cased and, when the warning filters are 'default' or 'always', they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using Python’s -Wd or -Wa options (see Warning control) and leaving `warnings` to `None`.

*Changed in version 3.2:* Added the `warnings` argument.

*Changed in version 3.2:* The default stream is set to `sys.stderr` at instantiation time rather than import time.

*Changed in version 3.5:* Added the `tb_locals` parameter.

_makeResult()

This method returns the instance of TestResult used by `run()`. It is not intended to be called directly, but can be overridden in subclasses to provide a custom TestResult.
_makeResult() instantiates the class or callable passed in the TextTestRunner constructor as the resultclass argument. It defaults to TextTestResult if no resultclass is provided. The result class is instantiated with the following arguments:

stream, descriptions, verbosity

run(test)
This method is the main public interface to the TextTestRunner. This method takes a TestSuite or TestCase instance. A TestResult is created by calling _makeResult() and the test(s) are run and the results printed to stdout.

unittest.main(module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None)

A command-line program that loads a set of tests from module and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

if __name__ == '__main__':
    unittest.main()

You can run tests with more detailed information by passing in the verbosity argument:

if __name__ == '__main__':
    unittest.main(verbosity=2)
The `defaultTest` argument is either the name of a single test or an iterable of test names to run if no test names are specified via `argv`. If not specified or `None` and no test names are provided via `argv`, all tests found in `module` are run.

The `argv` argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of `sys.argv` are used.

The `testRunner` argument can either be a test runner class or an already created instance of it. By default `main` calls `sys.exit()` with an exit code indicating success or failure of the tests run.

The `testLoader` argument has to be a `TestLoader` instance, and defaults to `defaultTestLoader`.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling `sys.exit()`:

```python
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The `failfast`, `catchbreak` and `buffer` parameters have the same effect as the same-name command-line options.

The `warnings` argument specifies the warning filter that should be used while running the tests. If it’s not specified, it will remain `None` if a `-W` option is passed to `python` (see Warning control), otherwise it will be set to `'default'`.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the result.
attribute.

*Changed in version 3.1:* The `exit` parameter was added.

*Changed in version 3.2:* The `verbosity`, `failfast`, `catchbreak`, `buffer` and `warnings` parameters were added.

*Changed in version 3.4:* The `defaultTest` parameter was changed to also accept an iterable of test names.

### load_tests Protocol

*New in version 3.2.*

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by `TestLoader.loadTestsFromModule()` with the following arguments:

```python
load_tests(loader, standard_tests, pattern)
```

where `pattern` is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a `TestSuite`.

*loader* is the instance of `TestLoader` doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.
A typical `load_tests` function that loads tests from a specific set of `TestCase` classes may look like:

```python
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling `TestLoader.discover()`, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package’s tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a `TestSuite` representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A ‘do nothing’ `load_tests` function for a test package would look like:

```python
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
```
```python
package_tests = loader.discover(start_dir=this_dir)
standard_tests.addTests(package_tests)
return standard_tests
```

*Changed in version 3.5:* Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.
Class and Module Fixtures

Class and module level fixtures are implemented in `TestSuite`. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don’t want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no
corresponding test instance an \_ErrorHolder object (that has the same interface as a testCase) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn’t matter, but if you are a framework author it may be relevant.

**setUpClass and tearDownClass**

These must be implemented as class methods:

```python
import unittest
class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the setUpClass and tearDownClass on base classes called then you must call up to them yourself. The implementations in testCase are empty.

If an exception is raised during a setUpClass then the tests in the class are not run and the tearDownClass is not run. Skipped classes will not have setUpClass or tearDownClass run. If the exception is a SkipTest exception then the class will be reported as having been skipped instead of as an error.

**setUpModule and tearDownModule**
These should be implemented as functions:

```python
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a `SkipTest` exception then the module will be reported as having been skipped instead of as an error.
Signal Handling

New in version 3.2.

The -c/--catch command-line option to unittest, along with the catchbreak parameter to unittest.main(), provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a KeyboardInterrupt in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own signal.SIGINT handler. If the unittest handler is called but isn't the installed signal.SIGINT handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need unittest control-c handling disabled the removeHandler() decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.**installHandler**()

Install the control-c handler. When a signal.SIGINT is received (usually in response to the user pressing control-c) all registered results have **stop**() called.

unittest.**registerResult**(result)
Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

`unittest.removeResult(result)`

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

`unittest.removeHandler(function=None)`

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```python
@unittest.removeHandler
def test_signal_handling(self):
    ...
```
unittest.mock — mock object library

New in version 3.3.

Source code: Lib/unittest/mock.py

unittest.mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

unittest.mock provides a core Mock class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a patch() decorator that handles patching module and class level attributes within the scope of a test, along with sentinel for creating unique objects. See the quick guide for some examples of how to use Mock, MagicMock and patch().

Mock is very easy to use and is designed for use with unittest. Mock is based on the ‘action -> assertion’ pattern instead of ‘record -> replay’ used by many mocking frameworks.

There is a backport of unittest.mock for earlier versions of Python, available as mock on PyPI.
Quick Guide

Mock and MagicMock objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```python
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect allows you to perform side effects, including raising an exception when a mock is called:

```python
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ... 
KeyError: 'foo'
```

```python
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```
Mock has many other ways you can configure it and control its behaviour. For example the `spec` argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don’t exist on the spec will fail with an `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```python
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
...   @patch('module.ClassName1')
...   def test(MockClass1, MockClass2):
...       module.ClassName1()
...       module.ClassName2()
...       assert MockClass1 is module.ClassName1
...       assert MockClass2 is module.ClassName2
...       assert MockClass1.called
...       assert MockClass2.called
...   >>> test()
```

**Note:** When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal Python order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

With `patch()` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read [where to patch](#).
As well as a decorator `patch()` can be used as a context manager in a `with` statement:

```python
>>> with patch.object(ProductionClass, 'method', return_value=...:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...     mock_method.assert_called_once_with(1, 2, 3)
```

There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```python
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...     assert foo == original
```

Mock supports the mocking of Python magic methods. The easiest way of using magic methods is with the `MagicMock` class. It allows you to do things like:

```python
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobaz'
>>> str(mock)
'foobaz'
>>> mock.__str__.assert_called_with()
```

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The `MagicMock` class is just a Mock variant that has all of the magic methods pre-
created for you (well, all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```python
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use auto-speccing. Auto-speccing can be done through the autospec argument to patch, or the create_autospec() function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```python
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...    pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```
create_autospec() can also be used on classes, where it copies the signature of the __init__ method, and on callable objects where it copies the signature of the __call__ method.
The Mock Class

Mock is a flexible mock object intended to replace the use of stubs and test doubles throughout your code.Mocks are callable and create attributes as new mocks when you access them [1]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

MagicMock is a subclass of Mock with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren’t callable: NonCallableMock and NonCallable MagicMock

The patch() decorators makes it easy to temporarily replace classes in a particular module with a Mock object. By default patch() will create a MagicMock for you. You can specify an alternative class of Mock using the new_callable argument to patch().

class unittest.mock.Mock(spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs)

Create a new Mock object. Mock takes several optional arguments that specify the behaviour of the Mock object:

- spec: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an AttributeError.
If `spec` is an object (rather than a list of strings) then `__class__` returns the class of the spec object. This allows mocks to pass `isinstance()` tests.

- **spec_set**: A stricter variant of `spec`. If used, attempting to set or get an attribute on the mock that isn't on the object passed as `spec_set` will raise an `AttributeError`.

- **side_effect**: A function to be called whenever the Mock is called. See the `side_effect` attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns `DEFAULT`, the return value of this function is used as the return value.

  Alternatively `side_effect` can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If `side_effect` is an iterable then each call to the mock will return the next value from the iterable.

  A `side_effect` can be cleared by setting it to `None`.

- **return_value**: The value returned when the mock is called. By default this is a new Mock (created on first access). See the `return_value` attribute.

- **unsafe**: By default if any attribute starts with `assert` or `assert` will raise an `AttributeError`. Passing `unsafe=True` will allow access to these attributes.

  *New in version 3.5.*
• wraps: Item for the mock object to wrap. If wraps is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn’t exist will raise an AttributeError).

If the mock has an explicit return_value set then calls are not passed to the wrapped object and the return_value is returned instead.

• name: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the configure_mock() method for details.

assert_called(*args, **kwargs)
Assert that the mock was called at least once.

```python
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method().assert_called()
```

New in version 3.6.

assert_called_once(*args, **kwargs)
Assert that the mock was called exactly once.

```python
>>> mock = Mock()
>>> mock.method()
```
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
  ...AssertionError: Expected 'method' to have been called once. Called 2 times.

New in version 3.6.

**assert_called_with(**args, **kwargs)**

This method is a convenient way of asserting that calls are made in a particular way:

```python
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

**assert_called_once_with(**args, **kwargs)**

Assert that the mock was called exactly once and that that call was with the specified arguments.

```python
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...AssertionError: Expected 'mock' to be called once on
```
**assert_any_call**(*args, **kwargs)

assert the mock has been called with the specified arguments.

The assert passes if the mock has ever been called, unlike **assert_called_with()** and **assert_called_once_with()** that only pass if the call is the most recent one, and in the case of **assert_called_once_with()** it must also be the only call.

```python
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

**assert_has_calls**(*calls, any_order=False)

assert the mock has been called with the specified calls. The **mock_calls** list is checked for the calls.

If **any_order** is false (the default) then the calls must be sequential. There can be extra calls before or after the specified calls.

If **any_order** is true then the calls can be in any order, but they must all appear in **mock_calls**.

```python
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```
assert_not_called()

Assert the mock was never called.

```python
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...  
AssertionError: Expected 'hello' to not have been called.
```

*New in version 3.5.*

reset_mock(*, return_value=False, side_effect=False)

The reset_mock method resets all the call attributes on a mock object:

```python
>>> mock = Mock(return_value=None)
>>> mock('hello')
True
>>> mock.called
False
>>> mock.reset_mock()
>>> mock.called
False
```

*Changed in version 3.6:* Added two keyword only argument to the reset_mock function.

This can be useful where you want to make a series of assertions that reuse the same object. Note that reset_mock() doesn't clear the return value, side_effect or any child attributes you have set using normal assignment by default. In case you want to reset return_value or
side_effect, then pass the corresponding parameter as True. Child mocks and the return value mock (if any) are reset as well.

Note: return_value, and side_effect are keyword only argument.

mock_add_spec(spec, spec_set=False)
Add a spec to a mock. spec can either be an object or a list of strings. Only attributes on the spec can be fetched as attributes from the mock.

If spec_set is true then only attributes on the spec can be set.

attach_mock(mock, attribute)
Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the method_calls and mock_calls attributes of this one.

configure_mock(**kwargs)
Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```python
generate_traceback
```
The same thing can be achieved in the constructor call to mocks:

```python
>>> attrs = {'method.return_value': 3, 'other.side_effect': None}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ... KeyError
```

configure_mock() exists to make it easier to do configuration after the mock has been created.

```python
__dir__()  
Mock objects limit the results of dir(some_mock) to useful results. For mocks with a spec this includes all the permitted attributes for the mock.

See FILTER_DIR for what this filtering does, and how to switch it off.
```

```python
_get_child_mock(**kw)
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.
```
For non-callable mocks the callable variant will be used (rather than any custom subclass).

**called**

A boolean representing whether or not the mock object has been called:

```python
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

**call_count**

An integer telling you how many times the mock object has been called:

```python
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

**return_value**

Set this to configure the value returned by calling the mock:

```python
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure
it in the normal way:

```python
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

`return_value` can also be set in the constructor:

```python
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

**side_effect**

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the `DEFAULT` singleton the call to the mock will then return whatever the function returns. If the function returns `DEFAULT` then the mock will return its normal value (from the `return_value`).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (`DEFAULT` handling is identical to the function case).
An example of a mock that raises an exception (to test exception handling of an API):

```python
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
 ...  
Exception: Boom!
```

Using `side_effect` to return a sequence of values:

```python
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Using a callable:

```python
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
... 
>>> mock.side_effect = side_effect
>>> mock()
3
```

`side_effect` can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```python
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
```
Setting `side_effect` to `None` clears it:

```python
>>> m = Mock(side_effect=KeyError, return_value=None)
>>> m()
Traceback (most recent call last):
...
KeyError
>>> m.side_effect = None
>>> m()
3
```

**call_args**

This is either `None` (if the mock hasn’t been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```python
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()  # call() results in call_args being set
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
```
```python
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

`call_args`, along with members of the lists `call_args_list`, `method_calls` and `mock_calls` are `call` objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See calls as tuples.

**call_args_list**

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The `call` object can be used for conveniently constructing lists of calls to compare with `call_args_list`.

```python
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
```

```python
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
```

```python
>>> expected = [(), ((3, 4),), ({'key': 'fish',
```

```python
>>> mock.call_args_list == expected
True
```

Members of `call_args_list` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See calls as tuples.

**method_calls**

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and their methods and attributes:
Mock

_mock = Mock()
_mock.method()
<Mock name='mock.method()' id='...'>
_mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>

mock.method_calls
[call.method(), call.property.method.attribute()]

Members of **method_calls** are **call** objects. These can be unpacked as tuples to get at the individual arguments. See calls as tuples.

mock_calls

_mock = MagicMock()
result = _mock(1, 2, 3)
_mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
_mock.second()
<MagicMock name='mock.second()' id='...'>
int(_mock)
1
result(1)
<MagicMock name='mock()' id='...'>
expected = [call(1, 2, 3), call.first(a=3), ...
call.__int__(), call().__int__(), call()(1)]
_mock.mock_calls == expected
True

Members of **mock_calls** are **call** objects. These can be unpacked as tuples to get at the individual arguments. See calls as tuples.
Note: The way `mock_calls` are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```python
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the `__class__` attribute of an object will return its type. For a mock object with a spec, `__class__` returns the spec class instead. This allows mock objects to pass `isinstance()` tests for the object they are replacing / masquerading as:

```python
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

`__class__` is assignable to, this allows a mock to pass an `isinstance()` check without forcing you to use a spec:

```python
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class `unittest.mock.NonCallableMock(spec=None,`
A non-callable version of `Mock`. The constructor parameters have the same meaning of `Mock`, with the exception of `return_value` and `side_effect` which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a spec or spec_set are able to pass `isinstance()` tests:

```python
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The `Mock` classes have support for mocking magic methods. See `magic methods` for the full details.

The mock classes and the `patch()` decorators all take arbitrary keyword arguments for configuration. For the `patch()` decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```python
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can’t use dotted names directly in a call you have to create a dictionary and unpack it using `**`:

```python
>>> attrs = {'method.return_value': 3, 'other.side_effect':

```
A callable mock which was created with a spec (or a spec_set) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```python
>>> def f(a, b, c): pass
... >>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to `assert_called_with()`, `assert_called_once_with()`, `assert_has_calls()` and `assert_any_call()`. When Autospeccing, it will also apply to method calls on the mock object.

Changed in version 3.4: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.PropertyMock(*args, **kwargs)

A mock intended to be used as a property, or other descriptor, on a class. PropertyMock provides `__get__()` and `__set__()`
methods so you can specify a return value when it is fetched.

Fetching a `PropertyMock` instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```python
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
... >>> with patch('__main__.Foo.foo', new_callable=PropertyMock):
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
... mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a `PropertyMock` to a mock object. Instead you can attach it to the mock type object:

```python
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```
Calling

Mock objects are callable. The call will return the value set as the `return_value` attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like `call_args` and `call_args_list`.

If `side_effect` is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make `side_effect` an exception class or instance:

```python
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'  
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]  
```

If `side_effect` is a function then whatever that function returns is
what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```python
global side_effect

>>> def side_effect(value):
...    return value + 1
...

>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return `DEFAULT`:

```python
global side_effect

>>> m = MagicMock()  # or any other mock
>>> def side_effect(*args, **kwargs):
...    return m.return_value
...

>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...    return DEFAULT
...

>>> m.side_effect = side_effect
>>> m()
3
```
To remove a side_effect, and return to the default behaviour, set the side_effect to None:

```python
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...    return 3
... >>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The side_effect can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a StopIteration is raised):

```python
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```python
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
```
Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return False to a `hasattr()` call, or raise an `AttributeError` when an attribute is fetched. You can do this by providing an object as a spec for a mock, but that isn’t always convenient.

You “block” attributes by deleting them. Once deleted, accessing an attribute will raise an `AttributeError`.

```python
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
  ...  
AttributeError: f
```
Mock names and the name attribute

Since “name” is an argument to the `Mock` constructor, if you want your mock object to have a “name” attribute you can’t just pass it in at creation time. There are two alternatives. One option is to use `configure_mock()`:

```python
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the “name” attribute after mock creation:

```python
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a “child” of that mock. Calls to the child are recorded in the `method_calls` and `mock_calls` attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```python
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
```
The exception to this is if the mock has a name. This allows you to prevent the “parenting” if for some reason you don’t want it to happen.

Mocks created for you by `patch()` are automatically given names. To attach mocks that have names to a parent you use the `attach_mock()` method:

```python
tests/test_05.py
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

The only exceptions are magic methods and attributes (those
that have leading and trailing double underscores). Mock doesn't create these but instead raises an `AttributeError`. [1] This is because the interpreter will often implicitly request these methods, and gets very confused to get a new Mock object when it expects a magic method. If you need magic method support see magic methods.
The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

**Note:** `patch()` is straightforward to use. The key is to do the patching in the right namespace. See the section *where to patch.*

`unittest.mock.patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

`patch()` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the `target` is patched with a `new` object. When the function/with statement exits the patch is undone.

If `new` is omitted, then the target is replaced with a `MagicMock`. If `patch()` is used as a decorator and `new` is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch()` is used as a context manager the created mock is returned by the context manager.

`target` should be a string in the form `'package.module.ClassName'`. The `target` is imported and the specified object replaced with the `new` object, so the `target` must be importable from the environment you are calling `patch()` from. The target is imported when the decorated function is executed,
not at decoration time.

The `spec` and `spec_set` keyword arguments are passed to the `MagicMock` if `patch` is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes `patch` to pass in the object being mocked as the `spec/spec_set` object.

`new_callable` allows you to specify a different class, or callable object, that will be called to create the `new` object. By default `MagicMock` is used.

A more powerful form of `spec` is `autospec`. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a `TypeError` if they are called with the wrong signature. For mocks replacing a class, their return value (the ‘instance’) will have the same spec as the class. See the `create_autospec()` function and `Autospeccing`.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch()` will fail to replace attributes that don’t exist. If you pass in `create=True`, and the attribute doesn’t exist, `patch` will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous.
With it switched on you can write passing tests against APIs that don't actually exist!

**Note:**

*Changed in version 3.5:* If you are patching builtins in a module then you don't need to pass create=True, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch()` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is 'test', which matches the way `unittest` finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use “as” then the patched object will be bound to the name after the “as”; very useful if `patch()` is creating a mock object for you.

`patch()` takes arbitrary keyword arguments. These will be passed to the `Mock` (or `new_callable`) on construction.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

`patch()` as function decorator, creating the mock for you and passing it into the decorated function:

```python
>>> @patch('__main__.SomeClass')
```
Patching a class replaces the class with a `MagicMock instance`. If the class is instantiated in the code under test then it will be the return_value of the mock that will be used.

If the class is instantiated multiple times you could use `side_effect` to return a new mock each time. Alternatively you can set the return_value to be anything you want.

To configure return values on methods of instances on the patched class you must do this on the return_value. For example:

```python
>>> class Class:
...     def method(self):
...         pass
... >>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
```

If you use `spec` or `spec_set` and `patch()` is replacing a class, then the return value of the created mock will have the same spec.

```python
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
```
The `new_callable` argument is useful where you want to use an alternative class to the default `MagicMock` for the created mock. For example, if you wanted a `NonCallableMock` to be used:

>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock)
...    assert thing is mock_thing
...    thing()
...
Traceback (most recent call last):
...
TypeError: 'NonCallableMock' object is not callable

Another use case might be to replace an object with an `io.StringIO` instance:

```python
>>> from io import StringIO
>>> def foo():
...    print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
...    def test(mock_stdout):
...        foo()
...        assert mock_stdout.getvalue() == 'Something'
...
>>> test()
```

When `patch()` is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords
you pass into the call will be used to set attributes on the created mock:

```python
generate
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch()` call using `**`:

```python
generate
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with `AttributeError`:

```python
generate
```
but adding `create=True` in the call to `patch()` will make the previous example work as expected:

```python
>>> @patch('sys.non_existing_attribute', 42, create=True)
...    def test(mock_stdout):
...        assert sys.non_existing_attribute == 42
...    
>>> test()
```

**patch.object**

`patch.object(target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)`

patch the named member (`attribute`) on an object (`target`) with a mock object.

`patch.object()` can be used as a decorator, class decorator or a context manager. Arguments `new, spec, create, spec_set, autospec` and `new_callable` have the same meaning as for `patch()`. Like `patch()`, `patch.object()` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.
You can either call `patch.object()` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```python
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
... >>> test()
```

`spec`, `create` and the other arguments to `patch.object()` have the same meaning as they do for `patch()`.

**patch.dict**

`patch.dict(in_dict, values=(), clear=False, **kwargs)`

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

`in_dict` can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

`in_dict` can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

`values` can be a dictionary of values to set in the dictionary. `values` can also be an iterable of `(key, value)` pairs.
If `clear` is true then the dictionary will be cleared before the new values are set.

`patch.dict()` can also be called with arbitrary keyword arguments to set values in the dictionary.

`patch.dict()` can be used as a context manager, decorator or class decorator. When used as a class decorator `patch.dict()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

`patch.dict()` can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```python
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```python
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...     newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the `patch.dict()` call to set values in the dictionary:

```python
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule)
```
`patch.dict()` can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```python
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...     assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

`patch.multiple`
patch.

\textbf{multiple}(target, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

\begin{verbatim}
  with patch.multiple(settings, FIRST_PATCH='one', ...
\end{verbatim}

Use DEFAULT as the value if you want patch.multiple() to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when patch.multiple() is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments spec, spec_set, create, autospec and new_callable have the same meaning as for patch(). These arguments will be applied to all patches done by patch.multiple().

When used as a class decorator patch.multiple() honours patch.TEST_PREFIX for choosing which methods to wrap.

If you want patch.multiple() to create mocks for you, then you can use DEFAULT as the value. If you use patch.multiple() as a decorator then the created mocks are passed into the decorated function by keyword.

\begin{verbatim}
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other
\end{verbatim}
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()

patch.multiple() can be nested with other patch decorators, but put arguments passed by keyword after any of the standard arguments created by patch():

>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()

If patch.multiple() is used as a context manager, the value returned by the context manger is a dictionary where created mocks are keyed by name:

>>> with patch.multiple('__main__', thing=DEFAULT, other
...     assert 'other' in repr(values['other']))
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other

patch methods: start and stop
All the patchers have start() and stop() methods. These make it simpler to do patching in setUp methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call patch(), patch.object() or patch.dict() as normal and keep a reference to the returned patcher object. You can then call start() to put the patch in place and stop() to undo it.

If you are using patch() to create a mock for you then it will be returned by the call to patcher.start.

```python
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the setUp method of a TestCase:

```python
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
```
Caution: If you use this technique you must ensure that the patching is “undone” by calling stop. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. `unittest.TestCase.addCleanup()` makes this easier:

As an added bonus you no longer need to keep a reference to the patcher object.

It is also possible to stop all patches which have been started by using `patch.stopall()`.

`patch.stopall()`

Stop all active patches. Only stops patches started with `start`.

`patch.builtins`
You can patch any builtins within a module. The following example patches builtin `ord()`:

```python
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

**TEST_PREFIX**

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `test` as being test methods. This is the same way that the `unittest.TestLoader` finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```python
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
```
Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```python
>>> @patch.object(SomeClass, 'class_method')
...   @patch.object(SomeClass, 'static_method')
...   def test(mock1, mock2):
...       assert SomeClass.static_method is mock1
...       assert SomeClass.class_method is mock2
...       SomeClass.static_method('foo')
...       SomeClass.class_method('bar')
...       return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

`patch()` works by (temporarily) changing the object that a `name`
points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```python
a.py
    -> Defines SomeClass
b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using `patch()`. The problem is that when we import module `b`, which we will have to do then it imports `SomeClass` from module `a`. If we use `patch()` to mock out `a.SomeClass` then it will have no effect on our test; module `b` already has a reference to the real `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module `b`, where we have imported it. The patching should look like:

```python
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a`
import SomeClass module b does import a and some_function uses a.SomeClass. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch a.SomeClass instead:

```python
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both `patch` and `patch.object` correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the `class` rather than an instance. They also work with `some` objects that proxy attribute access, like the `django settings object`. 
MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as “magic methods”. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods [2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes almost all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it must take self as the first argument [3].

```python
def __str__(self):
    ...    return 'fooble'
    ...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'

>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'

>>> mock = Mock()
```
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]

One use case for this is for mocking objects used as context managers in a `with` statement:

```python
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...    assert m == 'foo'
... ''
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in `method_calls`, but they are recorded in `mock_calls`.

**Note:** If you use the `spec` keyword argument to create a mock then attempting to set a magic method that isn’t in the spec will raise an `AttributeError`.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` and `__subclasses__`
- `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`
- Context manager: \_\_enter\_\_ and \_\_exit\_\_
- Unary numeric methods: \_\_neg\_\_, \_\_pos\_\_ and \_\_invert\_\_
- The numeric methods (including right hand and in-place variants): \_\_add\_\_, \_\_sub\_\_, \_\_mul\_\_, \_\_matmul\_\_, \_\_div\_\_, \_\_truediv\_\_, \_\_floordiv\_\_, \_\_mod\_\_, \_\_divmod\_\_, \_\_lshift\_\_, \_\_rshift\_\_, \_\_-\_\_and\_\_, \_\_-\_\_xor\_\_, \_\_-\_\_or\_\_, and \_\_-\_\_pow\_\_
- Numeric conversion methods: \_\_complex\_\_, \_\_int\_\_, \_\_float\_\_ and \_\_index\_\_
- Descriptor methods: \_\_get\_\_, \_\_set\_\_ and \_\_delete\_\_
- Pickling: \_\_reduce\_\_, \_\_reduce\_\_ex\_\_, \_\_getinitargs\_\_, \_\_getnewargs\_\_, \_\_getstate\_\_ and \_\_setstate\_\_

The following methods exist but are not supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- \_\_getattr\_\_, \_\_setattr\_\_, \_\_init\_\_ and \_\_new\_\_
- \_\_prepare\_\_, \_\_instancecheck\_\_, \_\_subclasscheck\_\_, \_\_del\_\_

**Magic Mock**

There are two MagicMock variants: MagicMock and NonCallableMagicMock.

```python
class unittest.mock.MagicMock(*args, **kw)
```

MagicMock is a subclass of Mock with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for Mock.
If you use the `spec` or `spec_set` arguments then only magic methods that exist in the spec will be created.

```python
class unittest.mock.NonCallableMagicMock(*args, **kw)
```
A non-callable version of `MagicMock`.

The constructor parameters have the same meaning as for `MagicMock`, with the exception of `return_value` and `side_effect` which have no meaning on a non-callable mock.

The magic methods are setup with `MagicMock` objects, so you can configure them and use them in the usual way:

```python
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren’t interested in the return value. You can still set the return value manually if you want to change the default.

Methods and their defaults:

- `__lt__`: NotImplemented
- `__gt__`: NotImplemented
- `__le__`: NotImplemented
- `__ge__`: NotImplemented
- `__int__`: 1
- `__contains__`: False
For example:

```python
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the `side_effect` attribute, unless you change their return value to return something else:

```python
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
```
The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator:

```python
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value is an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```python
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
• `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
• `__getformat__` and `__setformat__`

Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

The function is basically hooked up to the class, but each Mock instance is kept isolated from the others.
The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

*Changed in version 3.7:* The `sentinel` attributes now preserve their identity when they are copied or pickled.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. `sentinel` provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch method to return `sentinel.some_object`:

```python
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```
DEFAULT

unittest.mock.DEFAULT

The DEFAULT object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by side_effect functions to indicate that the normal return value should be used.

call

unittest.mock.call(*args, **kwargs)

call() is a helper object for making simpler assertions, for comparing with call_args, call_args_list, mock_calls and method_calls. call() can also be used with assert_has_calls().

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar')]
True
```

call.call_list()

For a call object that represents multiple calls, call_list() returns a list of all the intermediate calls as well as the final call.

call_list() is particularly useful for making assertions on “chained calls”. A chained call is multiple calls on a single line of code. This results in multiple entries in mock_calls on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same
chained call:

```python
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MockMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
call().method(arg='foo'),
call().method().other('bar'),
call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A `call` object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn’t particularly interesting, but the `call` objects that are in the `Mock.call_args`, `Mock.call_args_list` and `Mock.mock_calls` attributes can be introspected to get at the individual arguments they contain.

The `call` objects in `Mock.call_args` and `Mock.call_args_list` are two-tuples of (positional args, keyword args) whereas the `call` objects in `Mock.mock_calls`, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their “tupleness” to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```python
>>> m = MagicMock(return_value=None)
```
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True

>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True

create_autospec

unittest.mock.create_autospec(*spec, spec_set=False, instance=False, **kwargs)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the spec object as their spec.

Functions or methods being mocked will have their arguments
checked to ensure that they are called with the correct signature.

If spec_set is True then attempting to set attributes that don’t exist on the spec object will raise an AttributeError.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing instance=True. The returned mock will only be callable if instances of the mock are callable.

create_autospec() also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See Autospeccing for examples of how to use auto-speccing with create_autospec() and the autospec argument to patch().

ANY

unittest.mock.ANY

Sometimes you may need to make assertions about some of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of call_args and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to everything. Calls to assert_called_with() and assert_called_once_with() will then succeed no matter what was passed in.

```python
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
```
>>> mock.assert_called_once_with('foo', bar=ANY)

ANY can also be used in comparisons with call lists like mock_calls:

>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True

FILTER_DIR

unittest.mock.FILTER_DIR

FILTER_DIR is a module level variable that controls the way mock objects respond to dir() (only for Python 2.6 or more recent). The default is True, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set mock.FILTER_DIR = False.

With filtering on, dir(some_mock) shows only useful attributes and will include any dynamically created attributes that wouldn’t normally be shown. If the mock was created with a spec (or autospec of course) then all the attributes from the original are shown, even if they haven’t been accessed yet:

>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
```python
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler', 'AbstractHTTPHandler', 'BaseHandler', ...

Many of the not-very-useful (private to Mock rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling `dir()` on a Mock. If you dislike this behaviour you can switch it off by setting the module level switch `FILTER_DIR`:

```python
>>> from unittest import mock
>>> mock.FILTER_DIR = False

```python
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value', '_NonCallableMock__get_side_effect', '_NonCallableMock__return_value_doc', '_NonCallableMock__set_return_value', '_NonCallableMock__set_side_effect', '__call__', '__class__', ...

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of mock.FILTER_DIR.

mock_open

`unittest.mock.mock_open(mock=None, read_data=None)`

A helper function to create a mock to replace the use of `open()`. It
works for `open()` called directly or used as a context manager.

The `mock` argument is the mock object to configure. If `None` (the default) then a `MagicMock` will be created for you, with the API limited to methods or attributes available on standard file handles.

`read_data` is a string for the `read()`, `readline()`, and `readlines()` methods of the file handle to return. Calls to those methods will take data from `read_data` until it is depleted. The mock of these methods is pretty simplistic: every time the `mock` is called, the `read_data` is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

*Changed in version 3.4:* Added `readline()` and `readlines()` support. The mock of `read()` changed to consume `read_data` rather than returning it on each call.

*Changed in version 3.5:* `read_data` is now reset on each call to the `mock`.

*Changed in version 3.7.1:* Added `__iter__()` to implementation so that iteration (such as in for loops) correctly consumes `read_data`.

Using `open()` as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```python
with open('/some/path', 'w') as f:
    f.write('something')
```
The issue is that even if you mock out the call to `open()` it is the returned object that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a `MagicMock` is common enough and fiddly enough that a helper function is useful.

```python
>>> m = mock_open()
>>> with patch('__main__.open', m):
...    with open('foo', 'w') as h:
...        h.write('some stuff')

>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]

>>> m.assert_called_once_with('foo', 'w')

>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```python
>>> with patch('__main__.open', mock_open(read_data='bibble')) as h:
...    result = h.read()

>>> m.assert_called_once_with('foo')

>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing `spec` feature of mock. It limits
the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a **TypeError** if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

**Mock** is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the **Mock** api and the other is a more general problem with using mock objects.

First the problem specific to **Mock**. **Mock** has two assert methods that are extremely handy: **assert_called_with()** and **assert_called_once_with()**.

```python
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

```python
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(4, 5, 6)
```
Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the old api but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don’t test how your units are “wired together” there is still lots of room for bugs that tests might have caught.

mock already provides a feature to help with this, called speccing. If you use a class or instance as the spec for a mock then you can only access attributes on the mock that exist on the real class:

```python
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```python
>>> mock.has_data()
<MockMock object at 0x...>
>>> mock.has_data.assret_called_with()
```

Auto-speccing solves this problem. You can either pass autospec=True to patch() / patch.object() or use the
create_autospec() function to create a mock with a spec. If you use the autospec=True argument to patch() then the object that is being replaced will be used as the spec object. Because the speccing is done “lazily” (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here’s an example of it in use:

```python
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that request.Request has a spec. request.Request takes two arguments in the constructor (one of which is self). Here’s what happens if we try to call it incorrectly:

```python
>>> req = request.Request()
Traceback (most recent call last):
 ...  
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```python
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request'>
```
Request objects are not callable, so the return value of instantiating our mocked out request.Request is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```python
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assert_called_with
Traceback (most recent call last):
  ... AttributeError: Mock object has no attribute 'assert_called_with'
```

In many cases you will just be able to add `autospec=True` to your existing `patch()` calls and then be protected against bugs due to typos and api changes.

As well as using `autospec` through `patch()` there is a `create_autospec()` for creating autospecced mocks directly:

```python
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use
autospec. On the other hand it is much better to design your objects so that introspection is safe [4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. `autospec` can't know about any dynamically created attributes and restricts the api to visible attributes.

```python
>>> class Something:
...     def __init__(self):
...         self.a = 33
... >>> with patch('__main__.Something', autospec=True):
...     ...     thing = Something()
...     ...     thing.a
...     ...     Traceback (most recent call last):
...     ...     AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because `autospec` doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```python
>>> with patch('__main__.Something', autospec=True):
...     ...     thing = Something()
...     ...     thing.a = 33
...     ...     ...
```

There is a more aggressive version of both `spec` and `autospec` that does prevent you setting non-existent attributes. This is useful if you
want to ensure your code only sets valid attributes too, but obviously it prevents this particular scenario:

```python
>>> with patch('_main__.Something', autospec=True,
...    thing = Something()
...    thing.a = 33
... Traceback (most recent call last):
...    AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__() then providing them via class attributes (shared between instances of course) is faster too. e.g.

```python
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access any attributes or methods on it. As `None` is never going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```python
>>> class Something:
...    member = None
... >>> mock = create_autospec(Something)
```
If modifying your production classes to add defaults isn’t to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully `patch()` supports this - you can simply pass the alternative object as the `autospec` argument:

```python
>>> class Something:
    ...    def __init__(self):
        ...    self.a = 33
    ...
>>> class SomethingForTest(Something):
    ...    a = 33
    ...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallable MagicMock name='Something.a' spec='int'>
```

This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance does not create a real instance. It is only attribute lookups - along with calls to `dir()` - that are done.

Sealing mocks

```python
unittest.mock.seal(mock)
```

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that
are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won’t be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```python
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name"
>>> seal(mock)
>>> mock.new_attribute   # This will raise AttributeError
>>> mock.submock.attribute2   # This will raise AttributeError
>>> mock.not_submock.attribute2   # This won't raise.
```

New in version 3.7.
unittest.mock — getting started

New in version 3.3.
Using Mock

Mock Patching Methods

Common uses for Mock objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```python
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (real.method in this example) it has methods and attributes that allow you to make assertions about how it has been used.

**Note:** In most of these examples the Mock and MagicMock classes are interchangeable. As the MagicMock is the more capable class it makes a sensible one to use by default.

Once the mock has been called its `called` attribute is set to True. More importantly we can use the `assert_called_with()` or `assert_called_once_with()` method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the something method:
Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```python
>>> class ProductionClass:
    ...
    def closer(self, something):
        ...
        something.close()
    ...
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```python
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```
We don’t have to do any work to provide the ‘close’ method on our mock. Accessing close creates it. So, if ‘close’ hasn’t already been called then accessing it in the test will create it, but `assert_called_with()` will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by calling the class. This means you access the “mock instance” by looking at the return value of the mocked class.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to `patch()` replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock return value.

```python
>>> def some_function():
...    instance = module.Foo()
...    return instance.method()
...
>>> with patch('module.Foo') as mock:
...    instance = mock.return_value
...    instance.method.return_value = 'the result'
...    result = some_function()
...    assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the
repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```python
>>> mock = MagicMock(name='foo')
>>> mock
<MockMock name='foo' id='...'>
>>> mock.method
<MockMock name='foo.method' id='...'>
```

### Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```python
>>> mock = MagicMock()
>>> mock.method()
<MockMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MockMock name='mock.attribute.method(10, x=53)' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the `call` object to construct lists for comparing with `mock_calls`:
However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False)
True
```

### Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
```
If you need an attribute setting on your mock, just do it:

```python
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use `call` to construct the set of calls in a “chained call” like this for easy assertion afterwards:

```python
>>> mock = Mock()
>>> cursor = mock.connection.cursor().return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.execute.return_value = ['foo']
>>> expected = call.connection.cursor().execute("SELECT 1")
['foo']
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute("SELECT 1")]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

**Raising exceptions with mocks**
A useful attribute is `side_effect`. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```python
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...Exception: Boom!
```

**Side effect functions and iterables**

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```python
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```python
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
```
Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that also provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

`Mock` allows you to provide an object as a specification for the mock, using the `spec` keyword argument. Accessing methods / attributes on the mock that don’t exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as
positional or named arguments:

```python
>>> def f(a, b, c): pass
... >>> mock = Mock(spec=f)
... >>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
... >>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use auto-speccing.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use spec_set instead of spec.
Patch Decorators

**Note:** With `patch()` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: `patch()`, `patch.object()` and `patch.dict()`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. ‘patch.object’ takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

**patch.object:**

```python
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
>>> @patch('package.module.attribute', sentinel.attribute)
```
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
... >>> test()

If you are patching a module (including builtins) then use patch() instead of patch.object():

>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
... >>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"

The module name can be ‘dotted’, in the form package.module if needed:

>>> @patch('package.module.ClassName.attribute', sentinel)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
... >>> test()

A nice pattern is to actually decorate test methods themselves:

>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel)
If you want to patch with a Mock, you can use `patch()` with only one argument (or `patch.object()` with two arguments). The mock will be created for you and passed into the test function / method:

```python
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()

>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```python
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1,
...         self.assertIs(package.module.ClassName2,

>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.
There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```python
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...    assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use `patch()` to create a mock for you, you can get a reference to the mock using the “as” form of the with statement:

```python
>>> class ProductionClass:
...    ...
...    def method(self):
...        pass
...    ...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...    mock_method.return_value = None
...
>>> real = ProductionClass()
>>> real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with “test”.
Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the `return_value` attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new `Mock` is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```python
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this:

```python
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar'
```
Assuming that BackendProvider is already well tested, how do we test method()? Specifically, we want to test that the code section # more code uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the backend attribute on a Something instance. In this particular case we are only interested in the return value from the final call to start_call so we don't have much configuration to do. Let's assume the object it returns is ‘file-like’, so we'll ensure that our response object uses the builtin open() as its spec.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final start_call we could do this:

```python
mock_backend.get_endpoint.return_value.create_call.return_value.start_call
```

We can do that in a slightly nicer way using the configure_mock() method to directly set the return value for us:

```python
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call':
             mock_backend.configure_mock(**config)
```

With these we monkey patch the “mock backend” in place and can make the real call:

```python
>>> something.backend = mock_backend
```
Using `mock_calls` we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use `call.call_list()` to create this list of calls for us:

```python
>>> chained = call.get_endpoint('foobar').create_call
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to `datetime.date.today()` to return a known date, but I didn’t want to prevent the code under test from creating new date objects. Unfortunately `datetime.date` is written in C, and so I couldn’t just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The `patch` decorator is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```python
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10
```
Note that we don’t patch `datetime.date` globally, we patch `date` in the module that uses it. See where to patch.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over [1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is `__iter__()`, so we can mock this using a `MagicMock`.

Here’s an example class with an “iter” method implemented as a generator:
How would we mock this class, and in particular its “iter” method?

To configure the values returned from the iteration (implicit in the call to `list`), we need to configure the object returned by the call to `foo.iter()`.

```python
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

There are also generator expressions and more advanced uses of generators, but we aren’t concerned about them here. A very good introduction to generators and how powerful they are is: [Generator Tricks for Systems Programmers](#).

### Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use `patch()` (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:
An alternative way of managing patches is to use the patch methods: start and stop. These allow you to move the patching into your setUp and tearDown methods.
If you use this technique you must ensure that the patching is “undone” by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. `unittest.TestCase.addCleanup()` makes this easier:

```python
>>> class MyTest(TestCase):
...    def setUp(self):
...        patcher = patch('mymodule.foo')
...        self.addCleanup(patcher.stop)
...        self.mock_foo = patcher.start()
...    
...    def test_foo(self):
...        self.assertIs(mymodule.foo, self.mock_foo)

>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can’t patch with a mock for this, because if you replace an unbound method with a mock it doesn’t become a bound method when fetched from the instance, and so it doesn’t get self passed in. The workaround is to patch the unbound method with a real function instead. The `patch()` decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the
one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```python
>>> class Foo:
...     def foo(self):
...         pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...     foo = Foo()
...     foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don’t use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn’t called with `self`.

### Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```python
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the
assert_called_once_with() method that also asserts that the call_count is one.

```python
>>> mock.foo_bar.assert_called_once_with('baz', spam)
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam)
Traceback (most recent call last):
  ...  
AssertionError: Expected to be called once. Called 2
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use `call_args_list`:

```python
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The `call` helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```python
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

**Coping with mutable arguments**
Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store references to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in ‘mymodule’:

```python
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```python
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
... >>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
  ... AssertionError: Expected: (({6},), {}) Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.
Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using another mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```python
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

**Note:** If your mock is only going to be used once there is an easier
A way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```python
>>> def side_effect(arg):
...    assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
  ...  AssertionError
```

An alternative approach is to create a subclass of `Mock` or `MagicMock` that copies (using `copy.deepcopy()`) the arguments. Here's an example implementation:

```python
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...    def __call__(self, *args, **kwargs):
...        args = deepcopy(args)
...        kwargs = deepcopy(kwargs)
...        return super(CopyingMock, self).__call__(args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
  ...  AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
```
When you subclass Mock or MagicMock all dynamically created attributes, and the return_value will use your subclass automatically. That means all children of a CopyingMock will also have the type CopyingMock.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```python
>>> class MyTest(TestCase):
    ...
    ...    def test_foo(self):
    ...        with patch('mymodule.Foo') as mock_foo:
    ...            with patch('mymodule.Bar') as mock_bar:
    ...                with patch('mymodule.Spam') as mock_spam:
    ...                    assert mymodule.Foo is mock_foo
    ...                    assert mymodule.Bar is mock_bar
    ...                    assert mymodule.Spam is mock_spam
    ...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest cleanup functions and the patch methods: start and stop we can achieve the same effect without the nested indentation. A simple helper method, create_patch, puts the patch in place and returns the created mock for us:
Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with MagicMock, which will behave like a dictionary, and using side_effect to delegate dictionary access to a real underlying dictionary that is under our control.

When the __getitem__() and __setitem__() methods of our MagicMock are called (normal dictionary access) then side_effect is called with the key (and in the case of __setitem__ the value too). We can also control what is returned.
After the MagicMock has been used we can use attributes like `call_args_list` to assert about how the dictionary was used:

```python
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
    ...     return my_dict[name]
    ...
>>> def setitem(name, val):
    ...     my_dict[name] = val
    ...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

**Note:** An alternative to using MagicMock is to use Mock and only provide the magic methods you specifically want:

```python
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A third option is to use MagicMock but passing in `dict` as the `spec` (or `spec_set`) argument so that the MagicMock created only has dictionary magic methods available:

```python
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the mock will behave like a normal dictionary but recording the access. It even raises a `KeyError` if you try to access a key that doesn’t exist.
After it has been used you can make assertions about the access using the normal mock methods and attributes:

>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
  ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'

Mock subclasses and their attributes

There are various reasons why you might want to subclass Mock. One reason might be to add helper methods. Here's a silly example:

```python
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...```
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has Been_called()
False
>>> mymock()
>>> mymock.has BEEN Called()
True

The standard behaviour for Mock instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that Mock attributes are Mocks and MagicMock attributes are MagicMocks [2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has been_called()
True

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

Mock (in all its flavours) uses a method called _get_child_mock to create these “sub-mocks” for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (**kwargs) which are then passed onto the mock constructor:
```python
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
zMagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

An exception to this rule are the non-callable mocks. Attributes [2] use the callable variant because otherwise non-callable mocks couldn’t have callable methods.

## Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren’t using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is usually a much better way to solve the problem (refactor the code) or to prevent “up front costs” by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an `object` from the `sys.modules` dictionary. Note that it fetches an `object`, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.
This means you can use `patch.dict()` to *temporarily* put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```python
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock})
...    import fooble
...    fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in `sys.modules`.

This also works for the `from module import name` form:

```python
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock})
...    from fooble import blob
...    blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```python
>>> mock = Mock()
```
Tracking order of calls and less verbose call assertions

The Mock class allows you to track the order of method calls on your mock objects through the method_calls attribute. This doesn’t allow you to track the order of calls between separate mock objects, however we can use mock_calls to achieve the same effect.

Because mocks track calls to child mocks in mock_calls, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the mock_calls of the parent:

```python
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar

>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>

>>> manager.mock_calls
```
We can then assert about the calls, including the order, by comparing with the mock_calls attribute on the manager mock:

```python
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the `attach_mock()` method. After attaching calls will be recorded in `mock_calls` of the manager.

```python
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
  call.MockClass1().foo(),
  call.MockClass2(),
  call.MockClass2().bar()]
```

If many calls have been made, but you’re only interested in a particular sequence of them then an alternative is to use the `assert_has_calls()` method. This takes a list of calls (constructed with the `call` object). If that sequence of calls are in `mock_calls`

```python
[call.foo.something(), call.bar.other.thing()]```
then the assert succeeds.

```python
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m().one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about some of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```python
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as ANY we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use `assert_called_with()`
we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a ‘standard’ call to `assert_called_with` isn’t sufficient:

```python
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```python
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:
class Matcher:
... def __init__(self, compare, some_obj):
...     self.compare = compare
...     self.some_obj = some_obj
... def __eq__(self, other):
...     return self.compare(self.some_obj, other)

Putting all this together:

```python
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The Matcher is instantiated with our compare function and the Foo object we want to compare against. In `assert_called_with` the Matcher equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an `AssertionError` is raised:

```python
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
  ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the `AssertionError` directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality
matcher (hamcrest.library.integration.match_equality).
2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library `lib2to3` is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. `lib2to3` could also be adapted to custom applications in which Python code needs to be edited automatically.
Using 2to3

2to3 will usually be installed with the Python interpreter as a script. It is also located in the `Tools/scripts` directory of the Python root.

2to3’s basic arguments are a list of files or directories to transform. The directories are recursively traversed for Python sources.

Here is a sample Python 2.x source file, `example.py`:

```python
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless -n is also given.) Writing the changes back is enabled with the -w flag:

```
$ 2to3 -w example.py
```

After transformation, `example.py` looks like this:

```python
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
```
greet(name)

Comments and exact indentation are preserved throughout the translation process.

By default, 2to3 runs a set of predefined fixers. The -l flag lists all available fixers. An explicit set of fixers to run can be given with -f. Likewise the -x explicitly disables a fixer. The following example runs only the imports and has_key fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the apply fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are explicit, meaning they aren't run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the idioms fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

Notice how passing all enables all default fixers.

Sometimes 2to3 will find a place in your source code that needs to be changed, but 2to3 cannot fix automatically. In this case, 2to3 will print a warning beneath the diff for a file. You should address the warning in order to have compliant 3.x code.

2to3 can also refactor doctests. To enable this mode, use the -d flag. Note that only doctests will be refactored. This also doesn't require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.
The -v option enables output of more information on the translation process.

Since some print statements can be parsed as function calls or statements, 2to3 cannot always read files containing the print function. When 2to3 detects the presence of the `from __future__ import print_function` compiler directive, it modifies its internal grammar to interpret `print()` as a function. This change can also be enabled manually with the -p flag. Use -p to run fixers on code that already has had its print statements converted.

The -o or --output-dir option allows specification of an alternate directory for processed output files to be written to. The -n flag is required when using this as backup files do not make sense when not overwriting the input files.

*New in version 3.2.3:* The -o option was added.

The -W or --write-unchanged-files flag tells 2to3 to always write output files even if no changes were required to the file. This is most useful with -o so that an entire Python source tree is copied with translation from one directory to another. This option implies the -w flag as it would not make sense otherwise.

*New in version 3.2.3:* The -W flag was added.

The --add-suffix option specifies a string to append to all output filenames. The -n flag is required when specifying this as backups are not necessary when writing to different filenames. Example:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Will cause a converted file named `example.py3` to be written.
New in version 3.2.3: The `--add-suffix` option was added.

To translate an entire project from one directory tree to another use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```
Fixers

Each step of transforming code is encapsulated in a fixer. The command `2to3 -l` lists them. As documented above, each can be turned on and off individually. They are described here in more detail.

**apply**

Removes usage of `apply()`. For example `apply(function, *args, **kwargs)` is converted to `function(*args, **kwargs)`.

**asserts**

Replaces deprecated `unittest` method names with the correct ones.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>failUnlessEqual(a, b)</code></td>
<td><code>assertEqual(a, b)</code></td>
</tr>
<tr>
<td><code>assertEquals(a, b)</code></td>
<td><code>assertEqual(a, b)</code></td>
</tr>
<tr>
<td><code>failIfEqual(a, b)</code></td>
<td><code>assertNotEqual(a, b)</code></td>
</tr>
<tr>
<td><code>assertNotEquals(a, b)</code></td>
<td><code>assertNotEqual(a, b)</code></td>
</tr>
<tr>
<td><code>failUnless(a)</code></td>
<td><code>assertTrue(a)</code></td>
</tr>
<tr>
<td><code>assert_(a)</code></td>
<td><code>assertTrue(a)</code></td>
</tr>
<tr>
<td><code>failIf(a)</code></td>
<td><code>assertFalse(a)</code></td>
</tr>
<tr>
<td><code>failUnlessRaises(exc, cal)</code></td>
<td><code>assertRaises(exc, cal)</code></td>
</tr>
<tr>
<td><code>failUnlessAlmostEqual(a, b)</code></td>
<td><code>assertAlmostEqual(a, b)</code></td>
</tr>
<tr>
<td><code>assertAlmostEquals(a, b)</code></td>
<td><code>assertAlmostEqual(a, b)</code></td>
</tr>
</tbody>
</table>
failIfAlmostEqual(a, b) | assertNotAlmostEqual(a, b)
--- | ---
assertNotAlmostEquals(a, b) | assertNotAlmostEqual(a, b)

**basestring**
Converts basestring to *str*.

**buffer**
Converts buffer to *memoryview*. This fixer is optional because the *memoryview* API is similar but not exactly the same as that of buffer.

**dict**
Fixes dictionary iteration methods. dict.iteritems() is converted to *dict.items()*, dict.iterkeys() to *dict.keys()*, and dict.itervalues() to *dict.values()*. Similarly, dict.viewitems(), dict.viewkeys() and dict.viewvalues() are converted respectively to *dict.items()*, *dict.keys()* and *dict.values()*.

It also wraps existing usages of *dict.items()*, *dict.keys()* and *dict.values()* in a call to *list*.

**except**
Converts except *X, T* to except *X* as *T*.

**exec**
Converts the *exec* statement to the *exec()* function.

**execfile**
Removes usage of execfile(). The argument to execfile() is wrapped in calls to *open(), compile(),* and *exec()*.
exitfunc
Changes assignment of sys.exitfunc to use of the atexit module.

filter
Wraps filter() usage in a list call.

funcattrs
Fixes function attributes that have been renamed. For example,
my_function.func_closure is converted to
my_function.__closure__.

future
Removes from __future__ import new_feature statements.

getcwd
Renames os.getcwdu() to os.getcwd().

has_key
Changes dict.has_key(key) to key in dict.

idioms
This optional fixer performs several transformations that make
Python code more idiomatic. Type comparisons like type(x) is
SomeClass and type(x) == SomeClass are converted to
isinstance(x, SomeClass). while 1 becomes while True. This fixer also tries to make use of sorted() in appropriate
places. For example, this block

L = list(some_iterable)
L.sort()
is changed to

```python
L = sorted(some_iterable)
```

**import**

Detects sibling imports and converts them to relative imports.

**imports**

Handles module renames in the standard library.

**imports2**

Handles other modules renames in the standard library. It is separate from the *imports* fixer only because of technical limitations.

**input**

Converts `input(prompt)` to `eval(input(prompt))`.

**intern**

Converts `intern()` to `sys.intern()`.

**isinstance**

Fixes duplicate types in the second argument of `isinstance()`.
For example, `isinstance(x, (int, int))` is converted to `isinstance(x, int)` and `isinstance(x, (int, float, int))` is converted to `isinstance(x, (int, float))`.

**itertools_imports**

Removes imports of `itertools.ifilter()`, `itertools.izip()`, and `itertools.imap()`. Imports of `itertools.ifilterfalse()` are also changed to `itertools.filterfalse()`.
**itertools**
Changes usage of `itertools.ifilter()`, `itertools.izip()`, and `itertools.imap()` to their built-in equivalents. `itertools.ifilterfalse()` is changed to `itertools.filterfalse()`.

**long**
Renames `long` to `int`.

**map**
Wraps `map()` in a `list` call. It also changes `map(None, x)` to `list(x)`. Using `from future_builtins import map` disables this fixer.

**metaclass**
Converts the old metaclass syntax (`__metaclass__` = Meta in the class body) to the new (`class X(metaclass=Meta)`).

**methodattrs**
Fixes old method attribute names. For example, `meth.im_func` is converted to `meth.__func__`.

**ne**
Converts the old not-equal syntax, `<>`, to `!=`.

**next**
Converts the use of iterator’s `next()` methods to the `next()` function. It also renames `next()` methods to `__next__()`.

**nonzero**
Renames `__nonzero__()` to `__bool__()`.
numliterals
Converting octal literals into the new syntax.

operator
Converting calls to various functions in the `operator` module to other, but equivalent, function calls. When needed, the appropriate import statements are added, e.g. `import collections.abc`. The following mapping are made:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator.isCallable(obj)</code></td>
<td><code>callable(obj)</code></td>
</tr>
<tr>
<td><code>operator.sequenceIncludes(obj)</code></td>
<td><code>operator.contains(obj)</code></td>
</tr>
<tr>
<td><code>operator.isSequenceType(obj)</code></td>
<td><code>isinstance(obj, collections.abc.Sequence)</code></td>
</tr>
<tr>
<td><code>operator.isMappingType(obj)</code></td>
<td><code>isinstance(obj, collections.abc.Mapping)</code></td>
</tr>
<tr>
<td><code>operator.isNumberType(obj)</code></td>
<td><code>isinstance(obj, numbers.Number)</code></td>
</tr>
<tr>
<td><code>operator.repeat(obj, n)</code></td>
<td><code>operator.mul(obj, n)</code></td>
</tr>
<tr>
<td><code>operator.irepeat(obj, n)</code></td>
<td><code>operator.imul(obj, n)</code></td>
</tr>
</tbody>
</table>

paren
Add extra parenthesis where they are required in list comprehensions. For example, `[x for x in 1, 2]` becomes `[x for x in (1, 2)]`.

print
Converts the `print` statement to the `print()` function.

raise
Converts `raise E, V` to `raise E(V)`, and `raise E, V, T` to `raise E(V).with_traceback(T)`. If `E` is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in 3.0.

**raw_input**
Converts `raw_input()` to `input()`.

**reduce**
Handles the move of `reduce()` to `functools.reduce()`.

**reload**
Converts `reload()` to `importlib.reload()`.

**renames**
Changes `sys.maxint` to `sys.maxsize`.

**repr**
Replaces backtick `repr` with the `repr()` function.

**set_literal**
Replaces use of the `set` constructor with set literals. This fixer is optional.

**standarderror**
Renames `StandardError` to `Exception`.

**sys_exc**
Changes the deprecated `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` to use `sys.exc_info()`.

**throw**
Fixes the API change in generator's `throw()` method.
tuple_params
   Removes implicit tuple parameter unpacking. This fixer inserts temporary variables.

types
   Fixes code broken from the removal of some members in the types module.

unicode
   Renames unicode to str.

urllib
   Handles the rename of urllib and urllib2 to the urllib package.

ws_comma
   Removes excess whitespace from comma separated items. This fixer is optional.

xrange
   Renames xrange() to range() and wraps existing range() calls with list.

xreadlines
   Changes for x in file.xreadlines() to for x in file.

zip
   Wraps zip() usage in a list call. This is disabled when from future_builtins import zip appears.
lib2to3 - 2to3’s library

Source code: Lib/lib2to3/

Note: The lib2to3 API should be considered unstable and may change drastically in the future.
The `test` package contains all regression tests for Python as well as the modules `test.support` and `test.regrtest`. `test.support` is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the `test` package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the `unittest` or `doctest` module. Some older tests are written using a “traditional” testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

**See also:**

**Module unittest**
Writing PyUnit regression tests.

**Module doctest**
Tests embedded in documentation strings.
Writing Unit Tests for the `test` package

It is preferred that tests that use the `unittest` module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```python
import unittest
from test import support

class MyTestCase1(unittest.TestCase):
    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests

    def tearDown(self):
        ... code to execute to clean up after tests

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...
```
def test_feature_two(self):
    # Test feature two.
    ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

    ... more test classes ...

if __name__ == '__main__':
    unittest.main()

This code pattern allows the testing suite to be run by
test.regrtest, on its own as a script that supports the unittest
CLI, or via the python -m unittest CLI.

The goal for regression testing is to try to break code. This leads to a
few guidelines to be followed:

- The testing suite should exercise all classes, functions, and
  constants. This includes not just the external API that is to be
  presented to the outside world but also “private” code.

- Whitebox testing (examining the code being tested when the tests
  are being written) is preferred. Blackbox testing (testing only the
  published user interface) is not complete enough to make sure all
  boundary and edge cases are tested.

- Make sure all possible values are tested including invalid ones.
  This makes sure that not only all valid values are acceptable but
  also that improper values are handled correctly.
• Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.

• Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.

• Make sure to clean up after your tests (such as close and remove all temporary files).

• If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.

• Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.

• Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```python
class TestFuncAcceptsSequencesMixin:
    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin,
                arg = [1, 2, 3])
```
When using this pattern, remember that all classes that inherit from `unittest.TestCase` are run as tests. The Mixin class in the example above does not have any data and so can't be run by itself, thus it does not inherit from `unittest.TestCase`.

See also:

**Test Driven Development**
A book by Kent Beck on writing tests before code.
Running tests using the command-line interface

The test package can be run as a script to drive Python’s regression test suite, thanks to the -m option: python -m test. Under the hood, it uses test.regrtest; the call python -m test.regrtest used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the test package. It does this by finding all modules in the package whose name starts with test_, importing them, and executing the function test_main() if present or loading the tests via unittest.TestLoader.loadTestsFromModule if test_main does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (python -m test test_spam) will minimize output and only print whether the test passed or failed.

Running test directly allows what resources are available for tests to use to be set. You do this by using the -u command-line option. Specifying all as the value for the -u option enables all possible resources: python -m test -uall. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after all. The command python -m test -uall,-audio,-largefile will run test with all resources except the audio and largefile resources. For a list of all resources and more command-line options, run python -m test -h.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run make test at the top-level directory where Python was built. On Windows, executing rt.bat from your PCbuild directory will run all regression
tests.
test.support — Utilities for the Python test suite

The test.support module provides support for Python’s regression test suite.

**Note:** test.support is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

*exception* test.support.**TestFailed**

Exception to be raised when a test fails. This is deprecated in favor of unittest-based tests and unittest.TestCase’s assertion methods.

*exception* test.support.**ResourceDenied**

Subclass of unittest.SkipTest. Raised when a resource (such as a network connection) is not available. Raised by the requires() function.

The test.support module defines the following constants:

*test.support.**verbose**

True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. verbose is set by test.regrtest.
test.support.is_jython
    True if the running interpreter is Jython.

test.support.is_android
    True if the system is Android.

test.support.unix_shell
    Path for shell if not on Windows; otherwise None.

test.support.FS_NONASCII
    A non-ASCII character encodable by os.fsencode().

test.support.TESTFN
    Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.TESTFN_UNICODE
    Set to a non-ASCII name for a temporary file.

test.support.TESTFN_ENCODING
    Set to sys.getfilesystemencoding().

test.support.TESTFN_UNENCODABLE
    Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be None if it's not possible to generate such a filename.

test.support.TESTFN_UNDECODABLE
    Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be None if it's not possible to generate such a filename.
test.support.TESTFN_NONASCII
Set to a filename containing the FS_NONASCII character.

test.support.IPV6_ENABLED
Set to True if IPV6 is enabled on this host, False otherwise.

test.support.SAVEDCWD
Set to os.getcwd().

test.support.PGO
Set when tests can be skipped when they are not useful for PGO.

test.support.PIPE_MAX_SIZE
A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.SOCK_MAX_SIZE
A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.TEST_SUPPORT_DIR
Set to the top level directory that contains test.support.

test.support.TEST_HOME_DIR
Set to the top level directory for the test package.

test.support.TEST_DATA_DIR
Set to the data directory within the test package.

test.support.MAX_Py_ssize_t
Set to sys.maxsize for big memory tests.

test.support.max_memuse
Set by `set_memlimit()` as the memory limit for big memory tests. Limited by `MAX_Py_ssize_t`.

test.support.**real_max_memuse**
Set by `set_memlimit()` as the memory limit for big memory tests. Not limited by `MAX_Py_ssize_t`.

test.support.**MISSING_C_DOCSTRINGS**
Return `True` if running on CPython, not on Windows, and configuration not set with `WITH_DOC_STRINGS`.

test.support.**HAVE_DOCSTRINGS**
Check for presence of docstrings.

test.support.**TEST_HTTP_URL**
Define the URL of a dedicated HTTP server for the network tests.

The `test.support` module defines the following functions:

- **test.support.**`forget(module_name)`
  Remove the module named `module_name` from `sys.modules` and delete any byte-compiled files of the module.

- **test.support.**`unload(name)`
  Delete `name` from `sys.modules`.

- **test.support.**`unlink(filename)`
  Call `os.unlink()` on `filename`. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

- **test.support.**`rmdir(filename)`
  Call `os.rmdir()` on `filename`. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.
test.support.rmtree(path)
Call shutil.rmtree() on path or call os.lstat() and os.rmdir() to remove a path and its contents. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the files.

test.support.make_legacy_pyc(source)
Move a PEP 3147/488 pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The source value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

test.support.is_resource_enabled(resource)
Return True if resource is enabled and available. The list of available resources is only set when test.regrtest is executing the tests.

test.support.python_is_optimized()
Return True if Python was not built with -00 or -0g.

test.support.with_pymalloc()
Return _testcapi.WITH_PYMALLOC.

test.support.requires(resource, msg=None)
Raise ResourceDenied if resource is not available. msg is the argument to ResourceDenied if it is raised. Always returns True if called by a function whose __name__ is '__main__'. Used when tests are executed by test.regrtest.

test.support.system_must_validate_cert(f)
Raise unittest.SkipTest on TLS certification validation failures.
test.support.sortdict(dict)
    Return a repr of dict with keys sorted.

test.support.findfile(filename, subdir=None)
    Return the path to the file named filename. If no match is found
    filename is returned. This does not equal a failure since it could be
    the path to the file.

    Setting subdir indicates a relative path to use to find the file rather
    than looking directly in the path directories.

test.support.create_empty_file(filename)
    Create an empty file with filename. If it already exists, truncate it.

test.support.fd_count()
    Count the number of open file descriptors.

test.support.match_test(test)
    Match test to patterns set in set_match_tests().

test.support.set_match_tests(patterns)
    Define match test with regular expression patterns.

test.support.run_unittest(*classes)
    Execute unittest.TestCase subclasses passed to the function.
    The function scans the classes for methods starting with the prefix
    test_ and executes the tests individually.

    It is also legal to pass strings as parameters; these should be keys
    in sys.modules. Each associated module will be scanned by
    unittest.TestLoader.loadTestsFromModule(). This is
    usually seen in the following test_main() function:
This will run all tests defined in the named module.

def test_main():
    support.run_unittest(__name__)

test.support.run_doctest(module, verbosity=None, optionflags=0)
    Run doctest.testmod() on the given module. Return (failure_count, test_count).

    If verbosity is None, doctest.testmod() is run with verbosity set to verbose. Otherwise, it is run with verbosity set to None. optionflags is passed as optionflags to doctest.testmod().

test.support.setswitchinterval(interval)
    Set the sys.setswitchinterval() to the given interval. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.check_impl_detail(**guards)
    Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments:

    check_impl_detail() # Only on CPython
    check_impl_detail(jython=True) # Only on Jython
    check_impl_detail(cpython=False) # Everywhere except CPython

    test.support.check_warnings(*filters, quiet=True)
    A convenience wrapper for warnings.catch_warnings() that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling
warnings.catch_warnings(record=True) with

warnings.simplefilter() set to always and with the option to automatically validate the results that are recorded.

check_warnings accepts 2-tuples of the form ("message regexp", WarningCategory) as positional arguments. If one or more filters are provided, or if the optional keyword argument quiet is False, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set quiet to True.

If no arguments are specified, it defaults to:

```
check_warnings("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a WarningRecorder instance is returned. The underlying warnings list from catch_warnings() is available via the recorder object’s warnings attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return None.

The recorder object also has a reset() method, which clears the warnings list.

The context manager is designed to be used like this:
In this case if either warning was not raised, or some other warning was raised, `check_warnings()` would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```python
with check_warnings(("assertion is always true", "", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

*Changed in version 3.2:* New optional arguments `filters` and `quiet`.

test.support.**check_no_resource_warning**(testcase)

Context manager to check that no `ResourceWarning` was raised. You must remove the object which may emit `ResourceWarning` before the end of the context manager.

test.support.**set_memlimit**(limit)

Set the values for `max_memuse` and `real_max_memuse` for big
memory tests.

test.support.\texttt{record_original_stdout}(\texttt{stdout})

Store the value from \texttt{stdout}. It is meant to hold the stdout at the time the regrtest began.


test.support.\texttt{get_original_stdout}()

Return the original stdout set by \texttt{record_original_stdout()} or \texttt{sys.stdout} if it's not set.


test.support.\texttt{strip_python_strerr}(\texttt{stderr})

Strip the \texttt{stderr} of a Python process from potential debug output emitted by the interpreter. This will typically be run on the result of \texttt{subprocess.Popen.communicate()}.


test.support.\texttt{args_from_interpreter_flags}()

Return a list of command line arguments reproducing the current settings in \texttt{sys.flags} and \texttt{sys.warnoptions}.


test.support.\texttt{optim_args_from_interpreter_flags}()

Return a list of command line arguments reproducing the current optimization settings in \texttt{sys.flags}.


test.support.\texttt{captured_stdin}()
test.support.\texttt{captured_stdout}()
test.support.\texttt{captured_stderr}()

A context managers that temporarily replaces the named stream with \texttt{io.StringIO} object.

Example use with output streams:

\begin{verbatim}
with captured_stdout() as stdout, captured_stderr
  print("hello")
\end{verbatim}
print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"

Example use with input stream:

```python
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.temp_dir(path=None, quiet=False)

A context manager that creates a temporary directory at `path` and yields the directory.

If `path` is `None`, the temporary directory is created using `tempfile.mkdtemp()`. If `quiet` is `False`, the context manager raises an exception on error. Otherwise, if `path` is specified and cannot be created, only a warning is issued.

test.support.change_cwd(path, quiet=False)

A context manager that temporarily changes the current working directory to `path` and yields the directory.

If `quiet` is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.temp_cwd(name='tempcwd', quiet=False)

A context manager that temporarily creates a new directory and changes the current working directory (CWD).
The context manager creates a temporary directory in the current directory with name \textit{name} before temporarily changing the current working directory. If \textit{name} is \texttt{None}, the temporary directory is created using \texttt{tempfile.mkdtemp()}. If \textit{quiet} is \texttt{False} and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

\texttt{test.support.temp_umask}(\texttt{umask})

A context manager that temporarily sets the process umask.

\texttt{test.support.transient_internet}(\texttt{resource\_name}, *, \texttt{timeout}=30.0, \texttt{errnos}=())

A context manager that raises \texttt{ResourceDenied} when various issues with the internet connection manifest themselves as exceptions.

\texttt{test.support.disable_faulthandler}()

A context manager that replaces \texttt{sys.stderr} with \texttt{sys.__stderr__}.

\texttt{test.support.gc_collect}()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that \texttt{__del__} methods may be called later than expected and weakrefs may remain alive for longer than expected.

\texttt{test.support.disablegc}()

A context manager that disables the garbage collector upon entry and reenables it upon exit.
test.support.swap_attr(obj, attr, new_val)
Context manager to swap out an attribute with a new object.

Usage:

```python
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn’t exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn’t exist) will be assigned to the target of the “as” clause, if there is one.

test.support.swap_item(obj, attr, new_val)
Context manager to swap out an item with a new object.

Usage:

```python
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn’t exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn’t exist) will be assigned to the target of the “as” clause, if there is one.

test.support.wait_threads_exit(timeout=60.0)
Context manager to wait until all threads created in the `with` statement exit.
test.support.start_threads(threads, unlock=None)
    Context manager to start threads. It attempts to join the threads upon exit.

test.support.calcobjsize(fmt)
    Return struct.calcsize() for nP{fmt}0n or, if gettotalrefcount exists, 2PnP{fmt}0P.

test.support.calcvobjsize(fmt)
    Return struct.calcsize() for nPn{fmt}0n or, if gettotalrefcount exists, 2PnPn{fmt}0P.

test.support.checksizeof(test, o, size)
    For testcase test, assert that the sys.getsizeof for o plus the GC header size equals size.

test.support.can_symlink()
    Return True if the OS supports symbolic links, False otherwise.

test.support.can_xattr()
    Return True if the OS supports xattr, False otherwise.

@test.support.skip_unless_symlink
    A decorator for running tests that require support for symbolic links.

@test.support.skip_unless_xattr
    A decorator for running tests that require support for xattr.

@test.support.skip_unless_bind_unix_socket
    A decorator for running tests that require a functional bind() for Unix sockets.

@test.support.anticipate_failure(condition)
A decorator to conditionally mark tests with `unittest.expectedFailure()`. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

@test.support.run_with_locale(catstr, *locales)
A decorator for running a function in a different locale, correctly resetting it after it has finished. `catstr` is the locale category as a string (for example "LC_ALL"). The `locales` passed will be tried sequentially, and the first valid locale will be used.

@test.support.run_with_tz(tz)
A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.requires_freebsd_version(*min_version)
Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, raise `unittest.SkipTest`.

@test.support.requires_linux_version(*min_version)
Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, raise `unittest.SkipTest`.

@test.support.requires_mac_version(*min_version)
Decorator for the minimum version when running test on Mac OS X. If the MAC OS X version is less than the minimum, raise `unittest.SkipTest`.

@test.support.requires_IEEE_754
Decorator for skipping tests on non-IEEE 754 platforms.
@test.support.requires_zlib
Decorator for skipping tests if zlib doesn’t exist.

@test.support.requires_gzip
Decorator for skipping tests if gzip doesn’t exist.

@test.support.requires_bz2
Decorator for skipping tests if bz2 doesn’t exist.

@test.support.requires_lzma
Decorator for skipping tests if lzma doesn’t exist.

@test.supportQUIRES_RESOURCE(resource)
Decorator for skipping tests if resource is not available.

@test.support.requires_docstrings
Decorator for only running the test if HAVE_DOCSTRINGS.

@test.support.cpython_only(test)
Decorator for tests only applicable to CPython.

@test.support.impl_detail(msg=None, **guards)
Decorator for invoking check_impl_detail() on guards. If that returns False, then uses msg as the reason for skipping the test.

@test.support.no_tracing(func)
Decorator to temporarily turn off tracing for the duration of the test.

@test.support.refcount_test(test)
Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.
@test.support.reap_threads(func)
Decorator to ensure the threads are cleaned up even if the test fails.

@test.support.bigmemtest(size, memuse, dry_run=True)
Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) memuse is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with @bigmemtest(size=_4G, memuse=2).

The size argument is normally passed to the decorated test method as an extra argument. If dry_run is True, the value passed to the test method may be less than the requested value. If dry_run is False, it means the test doesn’t support dummy runs when -M is not specified.

@test.support.bigaddrspacetest(f)
Decorator for tests that fill the address space. f is the function to wrap.

test.support.make_bad_fd()
Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.check_syntax_error(testcase, statement, errtext='', *, lineno=None, offset=None)
Test for syntax errors in statement by attempting to compile statement. testcase is the unittest instance for the test. errtext is the text of the error raised by SyntaxError. If lineno is not
None, compares to the line of the `SyntaxError`. If `offset` is not None, compares to the offset of the `SyntaxError`.

test.support.**open_urlresource**(url, *args, **kw)
Open `url`. If open fails, raises `TestFailed`.

test.support.**import_module**(name, deprecated=False, *, required_on())
This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`. If a module is required on a platform but optional for others, set `required_on` to an iterable of platform prefixes which will be compared against `sys.platform`.

*New in version 3.1.*

test.support.**import_fresh_module**(name, fresh=(), blocked=(), deprecated=False)
This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

`fresh` is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

`blocked` is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise `ImportError`. 
The named module and any modules named in the `fresh` and `blocked` parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`.

This function will raise `ImportError` if the named module cannot be imported.

Example use:

```python
# Get copies of the warnings module for testing without affecting the version being used by the rest of the test suite. One copy uses the C implementation, the other is forced to use the pure Python fallback implementation
py_warnings = import_fresh_module('warnings', blocked)
c_warnings = import_fresh_module('warnings', fresh)
```

*New in version 3.1.*

`test.support.modules_setup()`  
Return a copy of `sys.modules`.

`test.support.modules_cleanup(oldmodules)`  
Remove modules except for `oldmodules` and `encodings` in order to preserve internal cache.

`test.support.threading_setup()`  
Return current thread count and copy of dangling threads.

`test.support.threading_cleanup(*original_values)`  
Cleanup up threads not specified in `original_values`. Designed to
emit a warning if a test leaves running threads in the background.

test.support.join_thread(thread, timeout=30.0)

Join a thread within timeout. Raise an AssertionError if thread is still alive after timeout seconds.

test.support.reap_children()

Use this at the end of test_main whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.get_attribute(obj, name)

Get an attribute, raising unittest.SkipTest if AttributeError is raised.

test.support.bind_port(sock, host=HOST)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the sock.family is AF_INET and sock.type is SOCK_STREAM, and the socket has SO_REUSEADDR or SO_REUSEPORT set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the SO_EXCLUSIVEADDRUSE socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.
test.support.bind_unix_socket(sock, addr)

Bind a unix socket, raising unittest.SkipTest if PermissionError is raised.

test.support.find_unused_port(family=socket.AF_INET, socktype=socket.SOCK_STREAM)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the sock parameter (default is AF_INET, SOCK_STREAM), and binding it to the specified host address (defaults to 0.0.0.0) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or bind_port() should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the -accept argument to openssl's s_server mode). Always prefer bind_port() over find_unused_port() where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.load_package_tests(pkg_dir, loader, standard_tests, pattern)

Generic implementation of the unittest load_tests protocol for use in test packages. pkg_dir is the root directory of the package; loader, standard_tests, and pattern are the arguments expected by load_tests. In simple cases, the test package's __init__.py can be the following:
```python
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__))
```

test.support.fs_is_case_insensitive(directory)
Return True if the file system for directory is case-insensitive.

test.support.detect_api_mismatch(ref_api, other_api, *, ignore=())
Returns the set of attributes, functions or methods of ref_api not found on other_api, except for a defined list of items to be ignored in this check specified in ignore.

By default this skips private attributes beginning with `_` but includes all magic methods, i.e. those starting and ending in `__`.

*New in version 3.5.*

test.support.patch(test_instance, object_to_patch, attr_name, new_value)
Override object_to_patch.attr_name with new_value. Also add cleanup procedure to test_instance to restore object_to_patch for attr_name. The attr_name should be a valid attribute for object_to_patch.

test.support.run_in_subinterp(code)
Run code in subinterpreter. Raise unittest.SkipTest if tracemalloc is enabled.

test.support.check_free_after_iterating(test, iter, cls, args=())
Assert that *iter* is deallocated after iterating.

test.support.missing_compiler_executable(*cmd_names*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or *None* when none is found missing.

test.support.check__all__(*test_case*, *module*, *name_of_module=None*, *extra=(), blacklist=())

Assert that the *__all__* variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like *csv* and its *_csv*).

The *extra* argument can be a set of names that wouldn’t otherwise be automatically detected as “public”, like objects without a proper *__module__* attribute. If provided, it will be added to the automatically detected ones.

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.
Example use:

```python
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', 'self'),
                             extra=extra, blacklist)
```

*New in version 3.6.*

The `test.support` module defines the following classes:

- **class test.support.TransientResource**(exc, **kwargs)
  Instances are a context manager that raises `ResourceDenied` if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is `ResourceDenied` raised.

- **class test.support.EnvironmentVarGuard**
  Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete
dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

*Changed in version 3.1:* Added dictionary interface.

```
EnvironmentVarGuard.set(envvar, value)
Temporarily set the environment variable envvar to the value of value.
```

```
EnvironmentVarGuard.unset(envvar)
Temporarily unset the environment variable envvar.
```

class test.support.SuppressCrashReport
A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetLastError.

On UNIX, resource.setrlimit() is used to set resource.RLIMIT_CORE’s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by __exit__().

class test.support.CleanImport(*module_names)
A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a DeprecationWarning on import. Example usage:
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.

class test.supportDirsOnSysPath(*paths)
    A context manager to temporarily add directories to sys.path.

    This makes a copy of sys.path, appends any directories given as positional arguments, then reverts sys.path to the copied settings when the context ends.

    Note that all sys.path modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

class test.supportSaveSignals
    Class to save and restore signal handlers registered by the Python signal handler.

class test.supportMatcher
    matches(self, d, **kwargs)
        Try to match a single dict with the supplied arguments.

    match_value(self, k, dv, v)
        Try to match a single stored value (dv) with a supplied value (v).

class test.supportWarningsRecorder
    Class used to record warnings for unit tests. See documentation of check_warnings() above for more details.

class test.supportBasicTestRunner
    run(test)
        Run test and return the result.
class test.support.TestHandler(logging.handlers.BufferingHandler)
    Class for logging support.

class test.support.FakePath(path)
    Simple path-like object. It implements the \_\_fspath\_\_( ) method which just returns the path argument. If path is an exception, it will be raised in \_\_fspath\_\_( ).
test.support.script_helper — Utilities for the Python execution tests

The test.support.script_helper module provides support for Python’s script execution tests.

test.support.script_helper.interpreter_requires_environment

Return True if sys.executable interpreter requires environment variables in order to be able to run at all.

This is designed to be used with @unittest.skipIf() to annotate tests that need to use an assert_python*() function to launch an isolated mode (-I) or no environment mode (-E) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn’t have an obvious home with Python’s current home finding logic.

Setting PYTHONHOME is one way to get most of the testsuite to run in that situation. PYTHONPATH or PYTHONUSERSITE are other common environment variables that might impact whether or not the interpreter can start.

test.support.script_helper.run_python_until_end(*args, **env_vars)

Set up the environment based on env_vars for running the interpreter in a subprocess. The values can include __isolated,
__cleanenv, __cwd, and TERM.

test.support.script_helper.assert_python_ok(*args, **env_vars)
    Assert that running the interpreter with args and optional environment variables env_vars succeeds (rc == 0) and return a (return code, stdout, stderr) tuple.

    If the __cleanenv keyword is set, env_vars is used as a fresh environment.

    Python is started in isolated mode (command line option -I), except if the __isolated keyword is set to False.

test.support.script_helper.assert_python_failure(**env_vars)
    Assert that running the interpreter with args and optional environment variables env_vars fails (rc != 0) and return a (return code, stdout, stderr) tuple.

    See assert_python_ok() for more options.

test.support.script_helper.spawn_python(*args, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, **kw)
    Run a Python subprocess with the given arguments.

    kw is extra keyword args to pass to subprocess.Popen(). Returns a subprocess.Popen object.

test.support.script_helper.kill_python(p)
    Run the given subprocess.Popen process until completion and return stdout.
test.support.script_helper.make_script(script_dir, script_basename, source, omit_suffix=False)

Create script containing source in path script_dir and script_basename. If omit_suffix is False, append .py to the name. Return the full script path.

test.support.script_helper.make_zip_script(zip_dir, zip_basename, script_name, name_in_zip=None)

Create zip file at zip_dir and zip_basename with extension zip which contains the files in script_name. name_in_zip is the archive name. Return a tuple containing (full path, full path of archive name).

test.support.script_helper.make_pkg(pkg_dir, init_source='')

Create a directory named pkg_dir containing an __init__ file with init_source as its contents.

test.support.script_helper.make_zip_pkg(zip_dir, zip_basename, pkg_name, script_basename, source, depth=1, compiled=False)

Create a zip package directory with a path of zip_dir and zip_basename containing an empty __init__ file and a file script_basename containing the source. If compiled is True, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.
Debugging and Profiling

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs.

- **bdb** — Debugger framework
- **faulthandler** — Dump the Python traceback
  - Dumping the traceback
  - Fault handler state
  - Dumping the tracebacks after a timeout
  - Dumping the traceback on a user signal
  - Issue with file descriptors
  - Example
- **pdb** — The Python Debugger
  - Debugger Commands
- **The Python Profilers**
  - Introduction to the profilers
  - Instant User’s Manual
  - *profile* and *cProfile* Module Reference
  - The Stats Class
  - What Is Deterministic Profiling?
  - Limitations
  - Calibration
  - Using a custom timer
- **timeit** — Measure execution time of small code snippets
  - Basic Examples
  - Python Interface
  - Command-Line Interface
  - Examples
- **trace** — Trace or track Python statement execution
  - Command-Line Usage
    - Main options
    - Modifiers
    - Filters
  - Programmatic Interface
- **tracemalloc** — Trace memory allocations
  - Examples
    - Display the top 10
    - Compute differences
    - Get the traceback of a memory block
    - Pretty top
  - API
    - Functions
    - DomainFilter
    - Filter
    - Frame
    - Snapshot
    - Statistic
    - StatisticDiff
    - Trace
    - Traceback
The `bdb` module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

```python
defense bdb.BdbQuit
    Exception raised by the Bdb class for quitting the debugger.
```

The `bdb` module also defines two classes:

```python
class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)
    This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.
```

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class `Breakpoint`. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a `funcname` is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

`Breakpoint` instances have the following methods:
**deleteMe()**
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

**enable()**
Mark the breakpoint as enabled.

**disable()**
Mark the breakpoint as disabled.

**bpformat()**
Return a string with all the information about the breakpoint, nicely formatted:

- The breakpoint number.
- If it is temporary or not.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

*New in version 3.2.*

**bpprint**(out=None)
Print the output of **bpformat()** to the file out, or if it is None, to standard output.

**class bdb . Bdb**(skip=None)
The **Bdb** class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (**pdb . Pdb**) is an example.
The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

*New in version 3.1:* The *skip* argument.

The following methods of *Bdb* normally don't need to be overridden.

**canonic**(*filename*)

Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

**reset**()

Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

**trace_dispatch**(*frame, event, arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to
return.

- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.
- "c_exception": A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The `arg` parameter depends on the previous event.

See the documentation for `sys.settrace()` for more information on the trace function. For more information on code and frame objects, refer to The standard type hierarchy.

**dispatch_line**(frame)

If the debugger should stop on the current line, invoke the `user_line()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the Bdb.quitting flag is set (which can be set from `user_line()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

**dispatch_call**(frame, arg)

If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the Bdb.quitting flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

**dispatch_return**(frame, arg)

If the debugger should stop on this function return, invoke the
user_return() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_return()). Return a reference to the trace_dispatch() method for further tracing in that scope.

dispatch_exception(frame, arg)

If the debugger should stop at this exception, invokes the user_exception() method (which should be overridden in subclasses). Raise a BdbQuit exception if the Bdb.quitting flag is set (which can be set from user_exception()). Return a reference to the trace_dispatch() method for further tracing in that scope.

Normally derived classes don’t override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(frame)

This method checks if the frame is somewhere below botframe in the call stack. botframe is the frame in which debugging started.

break_here(frame)

This method checks if there is a breakpoint in the filename and line belonging to frame or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(frame)

This method checks if there is a breakpoint in the filename of the current frame.
Derived classes should override these methods to gain control over debugger operation.

**user_call(frame, argument_list)**

This method is called from `dispatch_call()` when there is the possibility that a break might be necessary anywhere inside the called function.

**user_line(frame)**

This method is called from `dispatch_line()` when either `stop_here()` or `break_here()` yields True.

**user_return(frame, return_value)**

This method is called from `dispatch_return()` when `stop_here()` yields True.

**user_exception(frame, exc_info)**

This method is called from `dispatch_exception()` when `stop_here()` yields True.

**do_clear(arg)**

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

**set_step()**

Stop after one line of code.

**set_next(frame)**
Stop on the next line in or below the given frame.

**set_return(frame)**
Stop when returning from the given frame.

**set_until(frame)**
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

**set_trace(frame)**
Start debugging from frame. If frame is not specified, debugging starts from caller's frame.

**set_continue()**
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to None.

**set_quit()**
Set the quitting attribute to True. This raises BdbQuit in the next call to one of the dispatch_*() methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or None if all is well.

**set_break(filename, lineno, temporary=0, cond, funcname)**
Set a new breakpoint. If the lineno line doesn't exist for the filename passed as argument, return an error message. The filename should be in canonical form, as described in the canonic() method.

**clear_break(filename, lineno)**
Delete the breakpoints in filename and lineno. If none were set,
an error message is returned.

**clear_bpbynumber**(arg)
Delete the breakpoint which has the index arg in the Breakpoint.bpbynumber. If arg is not numeric or out of range, return an error message.

**clear_all_file_breaks**(filename)
Delete all breakpoints in filename. If none were set, an error message is returned.

**clear_all_breaks**()
Delete all existing breakpoints.

**get_bpbynumber**(arg)
Return a breakpoint specified by the given number. If arg is a string, it will be converted to a number. If arg is a non-numeric string, if the given breakpoint never existed or has been deleted, a *ValueError* is raised.

*New in version 3.2.*

**get_break**(filename, lineno)
Check if there is a breakpoint for lineno of filename.

**get_breaks**(filename, lineno)
Return all breakpoints for lineno in filename, or an empty list if none are set.

**get_file_breaks**(filename)
Return all breakpoints in filename, or an empty list if none are set.
get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(f, t)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(frame_lineno, lprefix=':')
Return a string with information about a stack entry, identified by a (frame, lineno) tuple:

- The canonical form of the filename which contains the frame.
- The function name, or "<lambda>".
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a statement, given as a string.

run(cmd, globals=None, locals=None)
Debug a statement executed via the exec() function. globals defaults to __main__.__dict__, locals defaults to globals.

runeval(expr, globals=None, locals=None)
Debug an expression executed via the eval() function. globals and locals have the same meaning as in run().

runctx(cmd, globals, locals)
For backwards compatibility. Calls the `run()` method.

```
runcall(func, *args, **kwds)
```
Debug a single function call, and return its result.

Finally, the module defines the following functions:

```
bdb.checkfuncname(b, frame)
```
Check whether we should break here, depending on the way the breakpoint `b` was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

```
bdb.effective(file, line, frame)
```
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

```
bdb.set_trace()
```
Start debugging with a `Bdb` instance from caller’s frame.
faulthandler — Dump the Python traceback

New in version 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call `faulthandler.enable()` to install fault handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS, and SIGILL signals. You can also enable them at startup by setting the `PYTHONFAULTHANDLER` environment variable or by using the `-X faulthandler` command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the `sigaltstack()` function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The `backslashreplace` error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed: the most recent call is shown first.
By default, the Python traceback is written to `sys.stderr`. To see
tracebacks, applications must be run in the terminal. A log file can
alternatively be passed to `faulthandler.enable()`.

The module is implemented in C, so tracebacks can be dumped on a
crash or when Python is deadlocked.
Dumping the traceback

\texttt{faulthandler.dump\_traceback} (\texttt{file=sys.stderr, all\_threads=True})

Dump the tracebacks of all threads into \texttt{file}. If \texttt{all\_threads} is \texttt{False}, dump only the current thread.

\textit{Changed in version 3.5}: Added support for passing file descriptor to this function.
Fault handler state

faulthandler.enable(file=sys.stderr, all_threads=True)

Enable the fault handler: install handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. If all_threads is True, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The file must be kept open until the fault handler is disabled: see issue with file descriptors.

*Changed in version 3.5:* Added support for passing file descriptor to this function.

*Changed in version 3.6:* On Windows, a handler for Windows exception is also installed.

faulthandler.disable()

Disable the fault handler: uninstall the signal handlers installed by enable().

faulthandler.is_enabled()

Check if the fault handler is enabled.
Dumping the tracebacks after a timeout

```
faulthandler.dump_traceback_later(timeout, repeat=False, file=sys.stderr, exit=False)
```

Dump the tracebacks of all threads, after a timeout of `timeout` seconds, or every `timeout` seconds if `repeat` is `True`. If `exit` is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn’t do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The `file` must be kept open until the traceback is dumped or `cancel_dump_traceback_later()` is called: see issue with file descriptors.

This function is implemented using a watchdog thread and therefore is not available if Python is compiled with threads disabled.

*Changed in version 3.5:* Added support for passing file descriptor to this function.

```
faulthandler.cancel_dump_traceback_later()
```

Cancel the last call to `dump_traceback_later()`.
Dumping the traceback on a user signal

```
faulthandler.register(signum, file=sys.stderr,
all_threads=True, chain=False)
```

Register a user signal: install a handler for the `signum` signal to
dump the traceback of all threads, or of the current thread if
`all_threads` is `False`, into `file`. Call the previous handler if `chain` is
`True`.

The `file` must be kept open until the signal is unregistered by
`unregister()`: see issue with file descriptors.

Not available on Windows.

*Changed in version 3.5:* Added support for passing file descriptor
to this function.

```
faulthandler.unregister(signum)
```

Unregister a user signal: uninstall the handler of the `signum` signal
installed by `register()`. Return `True` if the signal was
registered, `False` otherwise.

Not available on Windows.
Issue with file descriptors

`enable()`, `dump_traceback_later()` and `register()` keep the file descriptor of their `file` argument. If the file is closed and its file descriptor is reused by a new file, or if `os.dup2()` is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.
Example

Example of a segmentation fault on Linux with and without enabling the fault handler:

```bash
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault
```

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```
pdb — The Python Debugger

Source code: Lib/pdb.py

The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible – it is actually defined as the class Pdb. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules bdb and cmd.

The debugger’s prompt is (Pdb). Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Changed in version 3.3: Tab-completion via the readline module is available for commands and command arguments, e.g. the current
global and local names are offered as arguments of the p command.

`pdb.py` can also be invoked as a script to debug other scripts. For example:

```
python3 -m pdb myscript.py
```

When invoked as a script, `pdb` will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), `pdb` will restart the program. Automatic restarting preserves `pdb`’s state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program’s exit.

*New in version 3.2:* `pdb.py` now accepts a `-c` option that executes commands as if given in a `.pdbrc` file, see [Debugger Commands](#).

*New in version 3.7:* `pdb.py` now accepts a `-m` option that execute modules similar to the way `python3 -m` does. As with a script, the debugger will pause execution just before the first line of the module.

The typical usage to break into the debugger from a running program is to insert

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the `continue` command.

*New in version 3.7:* The built-in `breakpoint()`, when called with defaults, can be used instead of `import pdb; pdb.set_trace()`.
The typical usage to inspect a crashed program is:

```python
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/mymodule.py", line 4, in test
    test2()
  File "/mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

**pdb. run**(statement, globals=None, locals=None)

Execute the statement (given as a string or a code object) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type continue, or you can step through the statement using step or next (all these commands are explained below). The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module __main__ is used. (See the explanation of the built-in exec() or eval() functions.)

**pdb. runeval**(expression, globals=None, locals=None)

Evaluate the expression (given as a string or a code object) under debugger control. When runeval() returns, it returns the value of the expression. Otherwise this function is similar to run().
pdb. **runcall**(*function, *args, **kwds*)

Call the *function* (a function or method object, not a string) with the given arguments. When *runcall()* returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb. **set_trace**(*, header=None*)

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins.

*Changed in version 3.7:* The keyword-only argument *header*.

pdb. **post_mortem**(*traceback=None*)

Enter post-mortem debugging of the given *traceback* object. If no *traceback* is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

pdb. **pm**()

Enter post-mortem debugging of the traceback found in `sys.last_traceback`.

The *run* functions and *set_trace()* are aliases for instantiating the *Pdb* class and calling the method of the same name. If you want to access further features, you have to do this yourself:

```python
class pdb. **Pdb**(*completekey=’tab’, stdin=None, stdout=None, skip=None, nosigint=False, readrc=True*)

    *Pdb* is the debugger class.
```
The `completekey`, `stdin` and `stdout` arguments are passed to the underlying `cmd.Cmd` class; see the description there.

The `skip` argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. [1]

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses `ctrl-c` on the console) when you give a `continue` command. This allows you to break into the debugger again by pressing `ctrl-c`. If you want Pdb not to touch the SIGINT handler, set `nosigint` to true.

The `readrc` argument defaults to true and controls whether Pdb will load .pdbrc files from the filesystem.

Example call to enable tracing with `skip`:

```python
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

*New in version 3.1:* The `skip` argument.

*New in version 3.2:* The `nosigint` argument. Previously, a SIGINT handler was never set by Pdb.

*Changed in version 3.6:* The `readrc` argument.

- `run(statement, globals=None, locals=None)`
- `runeval(expression, globals=None, locals=None)`
- `runcall(function, *args, **kwds)`
- `set_trace()`

See the documentation for the functions explained above.
Debugger Commands

The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; e.g. h(elp) means that either h or help can be used to enter the help command (but not he or hel, nor H or Help or HELP). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([ ]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|).

Entering a blank line repeats the last command entered. Exception: if the last command was a list command, the next 11 lines are listed.

Commands that the debugger doesn’t recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (!). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger’s state is not changed.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

Multiple commands may be entered on a single line, separated by ;;. (A single ; is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ;; pair, even if it is in the middle of a quoted string.
If a file `.pdbrc` exists in the user’s home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

*Changed in version 3.2:* `.pdbrc` can now contain commands that continue debugging, such as `continue` or `next`. Previously, these commands had no effect.

**h(elp)** `[command]`
Without argument, print the list of available commands. With a `command` as argument, print help about that command. `help pdb` displays the full documentation (the docstring of the `pdb` module). Since the `command` argument must be an identifier, `help exec` must be entered to get help on the `!` command.

**w(here)**
Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

**d(own)** `[count]`
Move the current frame `count` (default one) levels down in the stack trace (to a newer frame).

**u(p)** `[count]`
Move the current frame `count` (default one) levels up in the stack trace (to an older frame).

**b(reak)** `[[filename:]lineno | function] [, condition]]`
With a `lineno` argument, set a break there in the current file. With a
function argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn’t been loaded yet). The file is searched on sys.path. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[[filename:]lineno | function] [, condition]]
Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as for break.

cl(ear) [[[filename:lineno | bpnumber [bpnumber ...]]]
With a filename:lineno argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [bpnumber [bpnumber ...]]
Disable the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable [bpnumber [bpnumber ...]]
Enable the breakpoints specified.

**ignore** bpnumber [count]
Set the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

**condition** bpnumber [condition]
Set a new condition for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

**commands** [bpnumber]
Specify a list of commands for breakpoint number *bpnumber*. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands. An example:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

With no `bpnumber` argument, `commands` refers to the last breakpoint set.

You can use breakpoint commands to start your program up again.
Simply use the `continue` command, or `step`, or any other command that resumes execution.

Specifying any command resuming execution (currently `continue`, `step`, `next`, `return`, `jump`, `quit` and their abbreviations) terminates the command list (as if that command was immediately followed by `end`). This is because any time you resume execution (even with a simple `next` or `step`), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

**s(step)**

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

**n(ext)**

Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

**unt(il) [lineno]**

Without argument, continue execution until the line with a number greater than the current one is reached.
With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

*Changed in version 3.2:* Allow giving an explicit line number.

**r(eturn)**
Continue execution until the current function returns.

**c(ont(inue))**
Continue execution, only stop when a breakpoint is encountered.

**j(ump) [lineno]**
Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don’t want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a *for* loop or out of a *finally* clause.

**l(ist) [first[, last]]**
List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With « as argument, list 11 lines around the current line. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

The current line in the current frame is indicated by «-». If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by «>>», if it differs from the current line.
New in version 3.2: The >> marker.

**ll** | longlist
List all source code for the current function or frame. Interesting lines are marked as for `list`.

New in version 3.2.

**a(rgs)**
Print the argument list of the current function.

**p** expression
Evaluate the *expression* in the current context and print its value.

**Note:** `print()` can also be used, but is not a debugger command — this executes the Python `print()` function.

**pp** expression
Like the *p* command, except the value of the expression is pretty-printed using the `pprint` module.

**whatis** expression
Print the type of the *expression*.

**source** expression
Try to get source code for the given object and display it.

New in version 3.2.

**display** [expression]
Display the value of the expression if it changed, each time execution stops in the current frame.
Without expression, list all display expressions for the current frame.

*New in version 3.2.*

**undisplay** [expression]

Do not display the expression any more in the current frame. Without expression, clear all display expressions for the current frame.

*New in version 3.2.*

**interact**

Start an interactive interpreter (using the `code` module) whose global namespace contains all the (global and local) names found in the current scope.

*New in version 3.2.*

**alias** [name [command]]

Create an alias called `name` that executes `command`. The command must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, and so on, while `%*` is replaced by all the parameters. If no command is given, the current alias for `name` is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands *can* be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when
placed in the .pdbrc file):

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.
# Print instance variables in self
alias ps pi self
```

**unalias** name

Delete the specified alias.

**! statement**

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a global statement on the same line, e.g.:

```
(Pdb) **global** list_options; list_options = ['-l']
(Pdb)
```

**run** [args ...]

**restart** [args ...]

Restart the debugged Python program. If an argument is supplied, it is split with shlex and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. **restart** is an alias for **run**.

**q**(uit)

Quit from the debugger. The program being executed is aborted.

**Footnotes**
Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.
The Python Profilers

Source code: Lib/profile.py and Lib/pstats.py
Introduction to the profilers

cProfile and profile provide deterministic profiling of Python programs. A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module.

The Python standard library provides two different implementations of the same profiling interface:

1. cProfile is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on lprof, contributed by Brett Rosen and Ted Czotter.
2. profile, a pure Python module whose interface is imitated by cProfile, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

**Note:** The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is timeit for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.
Instant User’s Manual

This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```python
import cProfile
import re
CProfile.run('re.compile("foo|bar")')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would run `re.compile()` and print profile results like the following:

```
197 function calls (192 primitive calls) in 0.
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno
1  0.000    0.000  0.001    0.001 <string>:1
1  0.000    0.000  0.001    0.001 re.py:212
1  0.000    0.000  0.001    0.001 re.py:268
1  0.000    0.000  0.000    0.000 sre_compile:1
1  0.000    0.000  0.000    0.000 sre_compile:1
4  0.000    0.000  0.000    0.000 sre_compile:3
3/1  0.000    0.000  0.000    0.000 sre_compile:3
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were `primitive`, meaning that the call was not induced via
recursion. The next line: **Ordered by: standard name**, indicates that the text string in the far right column was used to sort the output. The column headings include:

- **ncalls**
  for the number of calls.

- **tottime**
  for the total time spent in the given function (and excluding time made in calls to sub-functions)

- **percall**
  is the quotient of **tottime** divided by **ncalls**

- **cumtime**
  is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

- **percall**
  is the quotient of **cumtime** divided by primitive calls

- **filename:lineno(function)**
  provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function:

```python
import cProfile
import re
```
The `pstats.Stats` class reads profile results from a file and formats them in various ways.

The file `cProfile` can also be invoked as a script to profile another script. For example:

```python
python -m cProfile [-o output_file] [-s sort_order]
```

- `-o` writes the profile results to a file instead of to stdout
- `-s` specifies one of the `sort_stats()` sort values to sort the output by. This only applies when `-o` is not supplied.
- `-m` specifies that a module is being profiled instead of a script.

*New in version 3.7: Added the `-m` option.*

The `pstats` module's `Stats` class has a variety of methods for manipulating and printing the data saved into a profile results file:

```python
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The `strip_dirs()` method removed the extraneous path from all the module names. The `sort_stats()` method sorted all the entries according to the standard module/line/name string that is printed. The `print_stats()` method printed out all the statistics. You might try the following sort calls:
The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats()
```

This line sorts statistics with a primary key of time, and a secondary
key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the `pstats` module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using `cmd`) and interactive help.
profile and cProfile Module Reference

Both the profile and cProfile modules provide the following functions:

profile.run(command, filename=None, sort=-1)

This function takes a single argument that can be passed to the exec() function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.dict__, __main__.dict__
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a Stats instance and prints a simple profiling report. If the sort value is specified, it is passed to this Stats instance to control how the results are sorted.

profile.runctx(command, globals, locals, filename=None, sort=-1)

This function is similar to run(), with added arguments to supply the globals and locals dictionaries for the command string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the run() function above.

class profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)
This class is normally only used if more precise control over profiling is needed than what the cProfile.run() function provides.

A custom timer can be supplied for measuring how long code takes to run via the timer argument. This must be a function that returns a single number representing the current time. If the number is an integer, the timeunit specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be .001.

Directly using the Profile class allows formatting profile results without writing the profile data to a file:

```python
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
stream = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=stream).sort_stats(sortby)
ps.print_stats()
print(stream.getvalue())
```

**enable()**
Start collecting profiling data.

**disable()**
Stop collecting profiling data.

**create_stats()**
Stop collecting profiling data and record the results internally as the current profile.

**print_stats**(sort=-1)
Create a Stats object based on the current profile and print the results to stdout.

**dump_stats**(filename)
Write the results of the current profile to filename.

**run**(cmd)
Profile the cmd via **exec()**.

**runctx**(cmd, globals, locals)
Profile the cmd via **exec()** with the specified global and local environment.

**runcall**(func, *args, **kwargs)
Profile **func(*args, **kwargs)**

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a **sys.exit()** call during the called command/function execution) no profiling results will be printed.
The Stats Class

Analysis of the profiler data is done using the `Stats` class.

```python
class pstats.Stats(*filenames or profile, stream=sys.stdout)
```

This class constructor creates an instance of a “statistics object” from a `filename` (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by `stream`.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or `profile.Profile` object can be used as the profile data source.

`Stats` objects have the following methods:

**strip_dirs()**

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After
performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

**add(*filenames)**
This method of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()` or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

**dump_stats(filename)**
Save the data loaded into the Stats object to a file named `filename`. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the profile.Profile and cProfile.Profile classes.

**sort_stats(*keys)**
This method modifies the Stats object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: 'time', 'name', SortKey.TIME or SortKey.NAME). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.
When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, 
\[
sort\_stats(SortKey.NA\_ME, SortKey.FILE)\]
will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

<table>
<thead>
<tr>
<th>Valid String Arg</th>
<th>Valid enum Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'calls'</td>
<td>SortKey.CALLS</td>
<td>call count</td>
</tr>
<tr>
<td>'cumulative'</td>
<td>SortKey.CUMULATIVE</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'cumtime'</td>
<td>N/A</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'file'</td>
<td>N/A</td>
<td>file name</td>
</tr>
<tr>
<td>'filename'</td>
<td>SortKey.FILENAME</td>
<td>file name</td>
</tr>
<tr>
<td>'module'</td>
<td>N/A</td>
<td>file name</td>
</tr>
<tr>
<td>'ncalls'</td>
<td>N/A</td>
<td>call count</td>
</tr>
<tr>
<td>'pcalls'</td>
<td>SortKey.PCALLS</td>
<td>primitive call count</td>
</tr>
<tr>
<td>'line'</td>
<td>SortKey.LINE</td>
<td>line number</td>
</tr>
<tr>
<td>'name'</td>
<td>SortKey.NAME</td>
<td>function name</td>
</tr>
<tr>
<td>'nfl'</td>
<td>SortKey.NFL</td>
<td>name/file/line</td>
</tr>
<tr>
<td>'stdname'</td>
<td>SortKey.STDNAME</td>
<td>standard name</td>
</tr>
<tr>
<td>'time'</td>
<td>SortKey.TIME</td>
<td>internal time</td>
</tr>
</tbody>
</table>
'tottime'  N/A  internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

*New in version 3.7:* Added the `SortKey` enum.

**reverse_order()**
This method for the `Stats` class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

**print_stats(**`*restrictions`**)**
This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```python
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `*.foo:`. In contrast, the command:

```python
print_stats('foo:', .1)
```

would limit the list to all functions having file names `*.foo:`, and then proceed to only print the first 10% of them.

```python
print_callers(*restrictions)
```

This method for the `Stats` class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the
definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With **profile**, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With **cProfile**, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

**print_callees(*restrictions)**

This method for the **Stats** class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the **print_callers()** method.
What Is Deterministic Profiling?

_Deterministic profiling_ is meant to reflect the fact that all _function call_, _function return_, and _exception_ events are monitored, and precise timings are made for the intervals between these events (during which time the user’s code is executing). In contrast, _statistical profiling_ (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a _hook_ (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run-time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.
Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead cProfile. For this reason, profile provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). ) Do not be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually
better than without calibration.
Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see Limitations).

```python
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's `time.process_time()` as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is very fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```python
import profile

# 1. Apply computed bias to all Profile instances created thereafter.
profile.Profile.bias = your_computed_bias
```
# 2. Apply computed bias to a specific Profile instance.
```
pr = profile.Profile()
pr.bias = your_computed_bias
```

# 3. Specify computed bias in instance constructor.
```
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.
Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the Profile class constructor:

```python
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using `profile.Profile` or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

**profile.Profile**

`your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see `Calibration`). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is pretty bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

**cProfile.Profile**

`your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For
example, if your_integer_time_func returns times measured in thousands of seconds, you would construct the Profile instance as follows:

```python
pr = cProfile.Profile(your_integer_time_func, 0.0)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprowf` module.

Python 3.3 adds several new functions in `time` that can be used to make precise measurements of process or wall-clock time. For example, see `time.perf_counter()`.
timeit — Measure execution time of small code snippets

Source code: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the “Algorithms” chapter in the Python Cookbook, published by O'Reilly.
Basic Examples

The following example shows how the **Command-Line Interface** can be used to compare three different expressions:

```bash
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

This can be achieved from the **Python Interface** with:

```python
>>> import timeit
>>> timeit.timeit("""-".join(str(n) for n in range(100))""", 0.3018611848820001
>>> timeit.timeit("""-".join([str(n) for n in range(100)])""", 0.2727368790656328
>>> timeit.timeit("""-".join(map(str, range(100)))""", 0.23702679807320237
```

Note however that **timeit** will automatically determine the number of repetitions only when the command-line interface is used. In the **Examples** section you can find more advanced examples.
Python Interface

The module defines three convenience functions and a public class:

```python
timeit.timeit(stmt='pass', setup='pass', timer=<default timer>,
number=1000000, globals=None)
```
Create a Timer instance with the given statement, setup code and
timer function and run its timeit() method with number executions. The optional globals argument specifies a namespace in which to execute the code.

*Changed in version 3.5:* The optional globals parameter was added.

```python
timeit.repeat(stmt='pass', setup='pass', timer=<default timer>,
repeat=5, number=1000000, globals=None)
```
Create a Timer instance with the given statement, setup code and
timer function and run its repeat() method with the given repeat count and number executions. The optional globals argument specifies a namespace in which to execute the code.

*Changed in version 3.5:* The optional globals parameter was added.

*Changed in version 3.7:* Default value of repeat changed from 3 to 5.

```python
timeit.default_timer()
```
The default timer, which is always time.perf_counter().

*Changed in version 3.3:* time.perf_counter() is now the default timer.
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>, globals=None)

Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'; the timer function is platform-dependent (see the module doc string). stmt and setup may also contain multiple statements separated by ; or newlines, as long as they don't contain multi-line string literals. The statement will by default be executed within timeit's namespace; this behavior can be controlled by passing a namespace to globals.

To measure the execution time of the first statement, use the timeit() method. The repeat() and autorange() methods are convenience methods to call timeit() multiple times.

The execution time of setup is excluded from the overall timed execution run.

The stmt and setup parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by timeit(). Note that the timing overhead is a little larger in this case because of the extra function calls.

Changed in version 3.5: The optional globals parameter was added.

timeit(number=1000000)

Time number executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in
seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

Note: By default, \texttt{timeit()} temporarily turns off \texttt{garbage collection} during the timing. The advantage of this approach is that it makes independent timings more comparable. The disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the \texttt{setup} string. For example:

\begin{verbatim}
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()')
\end{verbatim}

\textbf{autorange}(\texttt{callback=None})

Automatically determine how many times to call \texttt{timeit()}. This is a convenience function that calls \texttt{timeit()} repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls \texttt{timeit()} with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, ... until the time taken is at least 0.2 second.

If \texttt{callback} is given and is not \texttt{None}, it will be called after each trial with two arguments: \texttt{callback(number, time_taken)}. New in version 3.6.

\textbf{repeat}(\texttt{repeat=5, number=1000000})

Call \texttt{timeit()} a few times.
This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the `number` argument for `timeit()`.

**Note:** It’s tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

*Changed in version 3.7:* Default value of `repeat` changed from 3 to 5.

**print_exc** *(file=None)*

Helper to print a traceback from the timed code.

Typical use:

```python
t = Timer(...)  # outside the try/except
try:
    t.timeit(...)  # or t.repeat(...)
except Exception:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional `file` argument directs where the traceback is sent; it defaults to
sys.stderr.
Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement]
```

Where the following options are understood:

- **-n N, --number=N**
  how many times to execute 'statement'

- **-r N, --repeat=N**
  how many times to repeat the timer (default 5)

- **-s S, --setup=S**
  statement to be executed once initially (default `pass`)

- **-p, --process**
  measure process time, not wallclock time, using `time.process_time()` instead of `time.perf_counter()`, which is the default

  *New in version 3.3.*

- **-u, --unit=U**
  specify a time unit for timer output; can select nsec, usec, msec, or sec

  *New in version 3.5.*

- **-v, --verbose**
print raw timing results; repeat for more digits precision

-h, --help
print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple -s options are treated similarly.

If -n is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

default_timer() measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The -r option is good for this; the default of 5 repetitions is probably enough in most cases. You can use time.process_time() to measure CPU time.

**Note:** There is a certain baseline overhead associated with executing a pass statement. The code here doesn’t try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.
Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```python
python -m timeit -s 'text = "sample string"; char = "g"'
```

$ python -m timeit -s 'text = "sample string"; char = "g"'
5000000 loops, best of 5: 0.0877 usec per loop

$ python -m timeit -s 'text = "sample string"; char = "g"'
1000000 loops, best of 5: 0.342 usec per loop

```python
>> import timeit
``` 

The same can be done using the `Timer` class and its methods:

```python
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.37866875250654886]
``` 

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using `hasattr()` vs. `try/except` to test for missing and present object attributes:

```python
python -m timeit 'try:' str.__bool__ ' except AttributeError:
```

$ python -m timeit 'try:' str.__bool__ ' except AttributeError:'
20000 loops, best of 5: 15.7 usec per loop

```python
python -m timeit 'if hasattr(str, "__bool__")':
```

$ python -m timeit 'if hasattr(str, "__bool__")':

50000 loops, best of 5: 4.26 usec per loop

```python
python -m timeit 'try:  		int.__bool__  
except AttributeError:  
' 200000 loops, best of 5: 1.43 usec per loop
python -m timeit 'if hasattr(int, '__bool__'): pass' 1000000 loops, best of 5: 2.23 usec per loop

```

To give the `timeit` module access to functions you define, you can pass a `setup` parameter which contains an import statement:
```python
def test():
    '''Stupid test function'''
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Another option is to pass `globals()` to the `globals` parameter, which will cause the code to be executed within your current global namespace. This can be more convenient than individually specifying imports:

```python
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]'))
```
trace — Trace or track Python statement execution

Source code: Lib/trace.py

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

See also:

**Coverage.py**

A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.
Command-Line Usage

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

---help
Display usage and exit.

--version
Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking `trace`. The `--listfuncs` option is mutually exclusive with the `--trace` and `--count` options. When `--listfuncs` is provided, neither `--count` nor `--trace` are accepted, and vice versa.

-c, --count
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also `--coverdir`, `--file` and `--no-report` below.

-t, --trace
Display lines as they are executed.
-l, --listfuncs
Display the functions executed by running the program.

-r, --report
Produce an annotated list from an earlier program run that used the --count and --file option. This does not execute any code.

-T, --trackcalls
Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
Name of a file to accumulate counts over several tracing runs. Should be used with the --count option.

-C, --coverdir=<dir>
Directory where the report files go. The coverage report for package.module is written to file dir/package/module.cover.

-m, --missing
When generating annotated listings, mark lines which were not executed with >>>>>>>.

-s, --summary
When using --count or --report, write a brief summary to stdout for each file processed.

-R, --no-report
Do not generate annotated listings. This is useful if you intend to
make several runs with --count, and then produce a single set of annotated listings at the end.

-g, --timing
Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>
Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>
Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by os.pathsep.
Programmatic Interface

class Trace(count=1, trace=1, countfuncs=0, countcallers=0, ignoremods=(), ignoredirs=(), infile=None, outfile=None, timing=False)

Create an object to trace execution of a single statement or expression. All parameters are optional. count enables counting of line numbers. trace enables line execution tracing. countfuncs enables listing of the functions called during the run. countcallers enables call relationship tracking. ignoremods is a list of modules or packages to ignore. ignoredirs is a list of directories whose modules or packages should be ignored. infile is the name of the file from which to read stored count information. outfile is the name of the file in which to write updated count information. timing enables a timestamp relative to when tracing was started to be displayed.

run(cmd)

Execute the command and gather statistics from the execution with the current tracing parameters. cmd must be a string or code object, suitable for passing into exec().

runctx(cmd, globals=None, locals=None)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, globals and locals default to empty dictionaries.

runfunc(func, *args, **kwds)

Call func with the given arguments under control of the Trace object with the current tracing parameters.
results()
    Return a CoverageResults object that contains the cumulative results of all previous calls to run, runctx and runfunc for the given Trace instance. Does not reset the accumulated trace results.

class trace.CoverageResults
    A container for coverage results, created by Trace.results(). Should not be created directly by the user.

    update(other)
        Merge in data from another CoverageResults object.

    write_results(show_missing=True, summary=False, coverdir=None)
        Write coverage results. Set show_missing to show lines that had no hits. Set summary to include in the output the coverage summary per module. coverdir specifies the directory into which the coverage result files will be output. If None, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

    import sys
    import trace

    # create a Trace object, telling it what to ignore, # do tracing or line-counting or both.
    tracer = trace.Trace(
        ignoredirs=[sys.prefix, sys.exec_prefix],
        trace=0,
```python
count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")```

---

**© Copyright** 2001-2019, Python Software Foundation. The Python Software Foundation is a non-profit corporation. [Please donate.](https://www.python.org/donate/)

Last updated on Mar 25, 2019. [Found a bug?](https://bugs.python.org)

Created using [Sphinx](http://sphinx.pocoo.org) 1.8.5.
tracemalloc — Trace memory allocations

New in version 3.4.

Source code: Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the PYTHONTRACEMALLOC environment variable to 1, or by using -X tracemalloc command line option. The tracemalloc.start() function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the PYTHONTRACEMALLOC environment variable to 25, or use the -X tracemalloc=25 command line option.
Examples

Display the top 10

Display the 10 files allocating the most memory:

```python
import tracemalloc
tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=1446, average=88.7 KiB,
<frozen importlib._bootstrap>:284: size=521 KiB, count=1454, average=52.0 KiB,
/usr/lib/python3.4/collections/__init__.py:368: size=716 KiB, count=1446, average=4855 KiB,
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=411, average=48.0 KiB,
```

We can see that Python loaded 4855 KiB data (bytecode and
constants) from modules and that the collections module allocated 244 KiB to build namedtuple types.

See Snapshot.statistics() for more options.

Compute differences

Take two snapshots and display the differences:

```python
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
```
We can see that Python has loaded 8173 KiB of module data (bytecode and constants), and that this is 4428 KiB more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the linecache module has cached 940 KiB of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the Snapshot.dump() method to analyze the snapshot offline. Then use the Snapshot.load() method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```python
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size))
for line in stat.traceback.format():
    print(line)
```
Example of output of the Python test suite (traceback limited to 25 frames):

903 memory blocks: 870.1 KiB
File "<frozen importlib._bootstrap>" , line 716
File "<frozen importlib._bootstrap>" , line 1036
File "<frozen importlib._bootstrap>" , line 934
File "<frozen importlib._bootstrap>" , line 1068
File "<frozen importlib._bootstrap>" , line 619
File "<frozen importlib._bootstrap>" , line 1581
File "<frozen importlib._bootstrap>" , line 1614
File "/usr/lib/python3.4/doctest.py" , line 101
    import pdb
File "<frozen importlib._bootstrap>" , line 284
File "<frozen importlib._bootstrap>" , line 938
File "<frozen importlib._bootstrap>" , line 1068
File "<frozen importlib._bootstrap>" , line 619
File "<frozen importlib._bootstrap>" , line 1581
File "<frozen importlib._bootstrap>" , line 1614
File "/usr/lib/python3.4/test/support/__init__.py" 
    import doctest
File "/usr/lib/python3.4/test/test_pickletools.py"
    support.run_doctest(pickletools)
File "/usr/lib/python3.4/test/regrtest.py" , line 1
    test_runner()
File "/usr/lib/python3.4/test/regrtest.py" , line 9
    display_failure=not verbose)
File "/usr/lib/python3.4/test/regrtest.py" , line 7
    match_tests=ns.match_tests)
File "/usr/lib/python3.4/test/regrtest.py" , line 1
    main()
File "/usr/lib/python3.4/test/__main__.py" , line 3
    regrtest.main_in_temp_cwd()
File "/usr/lib/python3.4/runpy.py" , line 73
exec(code, run_globals)
File "/usr/lib/python3.4/runpy.py" , line 160
    "__main__", fname, loader, pkg_name)
We can see that the most memory was allocated in the importlib
module to load data (bytecode and constants) from modules: 870.1
KiB. The traceback is where the importlib loaded data most
recently: on the import pdb line of the doctest module. The
traceback may change if a new module is loaded.

Pretty top
Code to display the 10 lines allocating the most memory with a pretty
output, ignoring <frozen
importlib._bootstrap>
and
<unknown> files:
import linecache
import os
import tracemalloc
def display_top(snapshot, key_type='lineno', limit=10
snapshot = snapshot.filter_traces((
tracemalloc.Filter(False, "<frozen importlib._bootst
tracemalloc.Filter(False, "<unknown>"),
))
top_stats = snapshot.statistics(key_type)
print("Top %s lines" % limit)
for index, stat in enumerate(top_stats[:limit],
frame = stat.traceback[0]
# replace "/path/to/module/file.py" with "module/fil
filename = os.sep.join(frame.filename.split(
print("#%s: %s:%s: %.1f KiB"
% (index, filename, frame.lineno, stat
line = linecache.getline(frame.filename, frame
if line:
print('
%s' % line)


other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)

Example of output of the Python test suite:

Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars2]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars2]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
\_b32tab2 = [a + b for a in \_b32tab for b in \_b32tab]

6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB

See `Snapshot.statistics()` for more options.
API

Functions

tracemalloc. **clear_traces()**
Clear traces of memory blocks allocated by Python.

See also **stop()**.

tracemalloc. **get_object_traceback(obj)**
Get the traceback where the Python object `obj` was allocated. Return a `Traceback` instance, or None if the `tracemalloc` module is not tracing memory allocations or did not trace the allocation of the object.

See also `gc.get_referrers()` and `sys.getsizeof()` functions.

tracemalloc. **get_traceback_limit()**
Get the maximum number of frames stored in the traceback of a trace.

The `tracemalloc` module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the `start()` function.

tracemalloc. **get_traced_memory()**
Get the current size and peak size of memory blocks traced by the `tracemalloc` module as a tuple: `(current: int, peak: int)`. 
tracemalloc. **get_tracemalloc_memory**()
Get the memory usage in bytes of the tracemalloc module used to store traces of memory blocks. Return an int.

tracemalloc. **is_tracing**()
True if the tracemalloc module is tracing Python memory allocations, False otherwise.

See also start() and stop() functions.

tracemalloc. **start**(nframe: int=1)
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to nframe frames. By default, a trace of a memory block only stores the most recent frame: the limit is 1. nframe must be greater or equal to 1.

Storing more than 1 frame is only useful to compute statistics grouped by 'traceback' or to compute cumulative statistics: see the Snapshot.compare_to() and Snapshot.statistics() methods.

Storing more frames increases the memory and CPU overhead of the tracemalloc module. Use the get_tracemalloc_memory() function to measure how much memory is used by the tracemalloc module.

The PYTHONTRACEMALLOC environment variable (PYTHONTRACEMALLOC=NFRAME) and the -X tracemalloc=NFRAME command line option can be used to start tracing at startup.
See also `stop()`, `is_tracing()` and `get_traceback_limit()` functions.

`tracemalloc.stop()`  
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call `take_snapshot()` function to take a snapshot of traces before clearing them.

See also `start()`, `is_tracing()` and `clear_traces()` functions.

`tracemalloc.take_snapshot()`  
Take a snapshot of traces of memory blocks allocated by Python. Return a new `Snapshot` instance.

The snapshot does not include memory blocks allocated before the `tracemalloc` module started to trace memory allocations.

Tracebacks of traces are limited to `get_traceback_limit()` frames. Use the `nframe` parameter of the `start()` function to store more frames.

The `tracemalloc` module must be tracing memory allocations to take a snapshot, see the `start()` function.

See also the `get_object_traceback()` function.

**DomainFilter**

```python
class tracemalloc.DomainFilter(inclusive: bool, domain: int)
```
Filter traces of memory blocks by their address space (domain).

*New in version 3.6.*

**inclusive**

If `inclusive` is `True` (include), match memory blocks allocated in the address space `domain`.

If `inclusive` is `False` (exclude), match memory blocks not allocated in the address space `domain`.

**domain**

Address space of a memory block (int). Read-only property.

**Filter**

class tracemalloc.Filter(inclusive: bool, filename_pattern: str, lineno: int=None, all_frames: bool=False, domain: int=None)

Filter on traces of memory blocks.

See the `fnmatch.fnmatch()` function for the syntax of `filename_pattern`. The `.pyc` file extension is replaced with `.py`.

Examples:

- `Filter(True, subprocess.__file__)` only includes traces of the `subprocess` module
- `Filter(False, tracemalloc.__file__)` excludes traces of the `tracemalloc` module
- `Filter(False, "<unknown>")` excludes empty tracebacks
Changed in version 3.5: The '.pyo' file extension is no longer replaced with '.py'.

Changed in version 3.6: Added the `domain` attribute.

**domain**
Address space of a memory block (int or None).

tracemalloc uses the domain 0 to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

**inclusive**
If `inclusive` is True (include), only match memory blocks allocated in a file with a name matching `filename_pattern` at line number `lineno`.

If `inclusive` is False (exclude), ignore memory blocks allocated in a file with a name matching `filename_pattern` at line number `lineno`.

**lineno**
Line number (int) of the filter. If `lineno` is None, the filter matches any line number.

**filename_pattern**
Filename pattern of the filter (str). Read-only property.

**all_frames**
If `all_frames` is True, all frames of the traceback are checked. If `all_frames` is False, only the most recent frame is checked.

This attribute has no effect if the traceback limit is 1. See the
get_traceback_limit() function and Snapshot.traceback_limit attribute.

Frame

class tracemalloc.Frame
    Frame of a traceback.

    The Traceback class is a sequence of Frame instances.

filename
    Filename (str).

lineno
    Line number (int).

Snapshot

class tracemalloc.Snapshot
    Snapshot of traces of memory blocks allocated by Python.

    The take_snapshot() function creates a snapshot instance.

compare_to(old_snapshot: Snapshot, key_type: str, cumulative: bool=False)
    Compute the differences with an old snapshot. Get statistics as a sorted list of StatisticDiff instances grouped by key_type.

    See the Snapshot.statistics() method for key_type and cumulative parameters.
The result is sorted from the biggest to the smallest by: absolute value of \texttt{StatisticDiff.size\_diff}, \texttt{StatisticDiff.size}, absolute value of \texttt{StatisticDiff.count\_diff}, \texttt{Statistic.count} and then by \texttt{StatisticDiff.traceback}.

\texttt{dump(filename)}

Write the snapshot into a file.

Use \texttt{load()} to reload the snapshot.

\texttt{filter\_traces(filters)}

Create a new \texttt{Snapshot} instance with a filtered \texttt{traces} sequence, \texttt{filters} is a list of \texttt{DomainFilter} and \texttt{Filter} instances. If \texttt{filters} is an empty list, return a new \texttt{Snapshot} instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

\textit{Changed in version 3.6} \texttt{DomainFilter} instances are now also accepted in \texttt{filters}.

\texttt{classmethod load(filename)}

Load a snapshot from a file.

See also \texttt{dump()}.

\texttt{statistics(key\_type: str, cumulative: bool=False)}

Get statistics as a sorted list of \texttt{Statistic} instances grouped by \texttt{key\_type}:
If `cumulative` is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with `key_type` equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: `Statistic.size`, `Statistic.count` and then by `Statistic.traceback`.

**traceback_limit**

Maximum number of frames stored in the traceback of traces: result of the `get_traceback_limit()` when the snapshot was taken.

**traces**

Traces of all memory blocks allocated by Python: sequence of `Trace` instances.

The sequence has an undefined order. Use the `Snapshot.statistics()` method to get a sorted list of statistics.

**Statistic**

class tracemalloc.statistic

Statistic on memory allocations.
`Snapshot.statistics()` returns a list of `Statistic` instances.

See also the `StatisticDiff` class.

**count**
Number of memory blocks (`int`).

**size**
Total size of memory blocks in bytes (`int`).

**traceback**
Traceback where the memory block was allocated, `Traceback` instance.

### StatisticDiff

*class* `tracemalloc.StatisticDiff`

Statistic difference on memory allocations between an old and a new `Snapshot` instance.

`Snapshot.compare_to()` returns a list of `StatisticDiff` instances. See also the `Statistic` class.

**count**
Number of memory blocks in the new snapshot (`int`): 0 if the memory blocks have been released in the new snapshot.

**count_diff**
Difference of number of memory blocks between the old and the new snapshots (`int`): 0 if the memory blocks have been allocated in the new snapshot.
**size**

Total size of memory blocks in bytes in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

**size_diff**

Difference of total size of memory blocks in bytes between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

**traceback**

Traceback where the memory blocks were allocated, *Traceback* instance.

---

**Trace**

`class tracemalloc.Trace`

Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

*Changed in version 3.6:* Added the *domain* attribute.

**domain**

Address space of a memory block (int). Read-only property.

tracemalloc uses the domain 0 to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

**size**
Size of the memory block in bytes (int).

**traceback**

Traceback where the memory block was allocated, `Traceback` instance.

**Traceback**

```python
class tracemalloc.Traceback
```

Sequence of `Frame` instances sorted from the oldest frame to the most recent frame.

A traceback contains at least 1 frame. If the `tracemalloc` module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to `get_traceback_limit()` frames. See the `take_snapshot()` function.

The `Trace.traceback` attribute is an instance of `Traceback` instance.

*Changed in version 3.7:* Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

**format(limit=None, most_recent_first=False)**

Format the traceback as a list of lines with newlines. Use the `linecache` module to retrieve lines from the source code. If `limit` is set, format the `limit` most recent frames if `limit` is positive. Otherwise, format the `abs(limit)` oldest frames. If `most_recent_first` is True, the order of the formatted frames is
reversed, returning the most recent frame first instead of last.

Similar to the `traceback.format_tb()` function, except that `format()` does not include newlines.

Example:

```python
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Output:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

© Copyright 2001-2019, Python Software Foundation. 
The Python Software Foundation is a non-profit corporation. Please donate. 
Last updated on Mar 25, 2019. Found a bug? 
Created using Sphinx 1.8.5.
Software Packaging and Distribution

These libraries help you with publishing and installing Python software. While these modules are designed to work in conjunction with the Python Package Index, they can also be used with a local index server, or without any index server at all.

- **distutils** — Building and installing Python modules
- **ensurepip** — Bootstrapping the pip installer
  - Command line interface
  - Module API
- **venv** — Creation of virtual environments
  - Creating virtual environments
  - API
  - An example of extending EnvBuilder
- **zipapp** — Manage executable Python zip archives
  - Basic Example
  - Command-Line Interface
  - Python API
  - Examples
  - Specifying the Interpreter
  - Creating Standalone Applications with zipapp
    - Making a Windows executable
    - Caveats
  - The Python Zip Application Archive Format
distutils — Building and installing Python modules

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

Most Python users will not want to use this module directly, but instead use the cross-version tools maintained by the Python Packaging Authority. In particular, setuptools is an enhanced alternative to distutils that provides:

- support for declaring project dependencies
- additional mechanisms for configuring which files to include in source releases (including plugins for integration with version control systems)
- the ability to declare project “entry points”, which can be used as the basis for application plugin systems
- the ability to automatically generate Windows command line executables at installation time rather than needing to prebuild them
- consistent behaviour across all supported Python versions

The recommended pip installer runs all setup.py scripts with setuptools, even if the script itself only imports distutils. Refer to the Python Packaging User Guide for more information.

For the benefits of packaging tool authors and users seeking a deeper understanding of the details of the current packaging and distribution
system, the legacy `distutils` based user documentation and API reference remain available:

- Installing Python Modules (Legacy version)
- Distributing Python Modules (Legacy version)
ensurepip — Bootstrapping the pip installer

New in version 3.4.

The ensurepip package provides support for bootstrapping the pip installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that pip is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn’t need to invoke this module directly (as pip should be bootstrapped by default), but it may be needed if installing pip was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling pip.

Note: This module does not access the internet. All of the components needed to bootstrap pip are included as internal parts of the package.

See also:

Installing Python Modules
The end user guide for installing Python packages

PEP 453: Explicit bootstrapping of pip in Python installations
The original rationale and specification for this module.
Command line interface

The command line interface is invoked using the interpreter’s `--m` switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install `pip` if it is not already installed, but otherwise does nothing. To ensure the installed version of `pip` is at least as recent as the one bundled with `ensurepip`, pass the `--upgrade` option:

```
python -m ensurepip --upgrade
```

By default, `pip` is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- `--root <dir>`: Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- `--user`: Installs `pip` into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts `pipX` and `pipX.Y` will be installed (where `X.Y` stands for the version of Python used to invoke `ensurepip`). The scripts installed can be controlled through two additional command line options:
- **--altinstall**: if an alternate installation is requested, the pipX script will *not* be installed.
- **--default-pip**: if a “default pip” installation is requested, the pip script will be installed in addition to the two regular scripts.

Providing both of the script selection options will trigger an exception.
Module API

**ensurepip** exposes two functions for programmatic use:

**ensurepip.version()**

Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

**ensurepip.bootstrap(root=None, upgrade=False, user=False, altinstall=False, default_pip=False, verbosity=0)**

Bootstraps pip into the current or designated environment.

*root* specifies an alternative root directory to install relative to. If *root* is *None*, then installation uses the default install location for the current environment.

*upgrade* indicates whether or not to upgrade an existing installation of an earlier version of pip to the bundled version.

*user* indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then `pipX` will *not* be installed.

If *default_pip* is set, then *pip* will be installed in addition to the two regular scripts.

Setting both *altinstall* and *default_pip* will trigger `ValueError`.

*verbosity* controls the level of output to `sys.stdout` from the
bootstrapping operation.

**Note:** The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

**Note:** The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).
**venv — Creation of virtual environments**

*New in version 3.3.*

**Source code:** `Lib/venv/`

The `venv` module provides support for creating lightweight “virtual environments” with their own site directories, optionally isolated from system site directories. Each virtual environment has its own Python binary (which matches the version of the binary that was used to create this environment) and can have its own independent set of installed Python packages in its site directories.

See [PEP 405](https://www.python.org/dev/peps/pep-0405/) for more information about Python virtual environments.

**See also:** [Python Packaging User Guide: Creating and using virtual environments](https://docs.python.org/3/distutils/)

**Note:** The `pyvenv` script has been deprecated as of Python 3.6 in favor of using `python3 -m venv` to help prevent any potential confusion as to which Python interpreter a virtual environment will be based on.
Creating virtual environments

Creation of virtual environments is done by executing the command `venv`:

```bash
python3 -m venv /path/to/new/virtual/environment
```

Running this command creates the target directory (creating any parent directories that don’t exist already) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run. It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Lib\site-packages`). If an existing directory is specified, it will be re-used.

*Deprecated since version 3.6:* `pyvenv` was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

*Changed in version 3.5:* The use of `venv` is now recommended for creating virtual environments.

On Windows, invoke the `venv` command as follows:

```bash
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Alternatively, if you configured the `PATH` and `PATHEXT` variables for your Python installation:
c:\>python -m venv c:\path\to\myenv

The command, if run with `-h`, will show the available options:

```

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR                        A directory to create the environment in.

optional arguments:
  -h, --help                      show this help message and exit
  --system-site-packages           Give the virtual environment access to the system site-packages dir.
  --symlinks                      Try to use symlinks rather than copies, even when symlinks are the default for the platform.
  --copies                        Try to use copies rather than symlinks, even when symlinks are the default for the platform.
  --clear                         Delete the contents of the environment directory if it already exists, before environment creation.
  --upgrade                       Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.
  --without-pip                   Skips installing or upgrading pip in the virtual environment (pip is bootstrapped by default).
  --prompt PROMPT                 Provides an alternative prompt prefix for this environment.

Once an environment has been created, you may wish to activate it, e.g., by sourcing an activate script in its bin directory.
```

*Changed in version 3.4:* Installs pip by default, added the `--without-pip` option.
without-pip and --copies options

*Changed in version 3.4:* In earlier versions, if the target directory already existed, an error was raised, unless the --clear or --upgrade option was provided.

**Note:** While symlinks are supported on Windows, they are not recommended. Of particular note is that double-clicking python.exe in File Explorer will resolve the symlink eagerly and ignore the virtual environment.

The created pyvenv.cfg file also includes the include-system-site-packages key, set to true if venv is run with the --system-site-packages option, false otherwise.

Unless the --without-pip option is given, ensurepip will be invoked to bootstrap pip into the virtual environment.

Multiple paths can be given to venv, in which case an identical virtual environment will be created, according to the given options, at each provided path.

Once a virtual environment has been created, it can be “activated” using a script in the virtual environment’s binary directory. The invocation of the script is platform-specific (<venv> must be replaced by the path of the directory containing the virtual environment):

<table>
<thead>
<tr>
<th>Platform</th>
<th>Shell</th>
<th>Command to activate virtual environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Posix</td>
<td>bash/zsh</td>
<td><code>$ source &lt;venv&gt;/bin/activate</code></td>
</tr>
<tr>
<td></td>
<td>fish</td>
<td><code>$. &lt;venv&gt;/bin/activate.fish</code></td>
</tr>
</tbody>
</table>
You don’t specifically need to activate an environment; activation just prepends the virtual environment’s binary directory to your path, so that “python” invokes the virtual environment’s Python interpreter and you can run installed scripts without having to use their full path. However, all scripts installed in a virtual environment should be runnable without activating it, and run with the virtual environment’s Python automatically.

You can deactivate a virtual environment by typing “deactivate” in your shell. The exact mechanism is platform-specific: for example, the Bash activation script defines a “deactivate” function, whereas on Windows there are separate scripts called deactivate.bat and Deactivate.ps1 which are installed when the virtual environment is created.

New in version 3.4: fish and csh activation scripts.

---

**Note:** A virtual environment is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments, and (by default) any libraries installed in a “system” Python, i.e., one which is installed as part of your operating system.

A virtual environment is a directory tree which contains Python executable files and other files which indicate that it is a virtual environment.

Common installation tools such as setuptools and pip work as
expected with virtual environments. In other words, when a virtual environment is active, they install Python packages into the virtual environment without needing to be told to do so explicitly.

When a virtual environment is active (i.e., the virtual environment’s Python interpreter is running), the attributes `sys.prefix` and `sys.exec_prefix` point to the base directory of the virtual environment, whereas `sys.base_prefix` and `sys.base_exec_prefix` point to the non-virtual environment Python installation which was used to create the virtual environment. If a virtual environment is not active, then `sys.prefix` is the same as `sys.base_prefix` and `sys.exec_prefix` is the same as `sys.base_exec_prefix` (they all point to a non-virtual environment Python installation).

When a virtual environment is active, any options that change the installation path will be ignored from all distutils configuration files to prevent projects being inadvertently installed outside of the virtual environment.

When working in a command shell, users can make a virtual environment active by running an `activate` script in the virtual environment’s executables directory (the precise filename is shell-dependent), which prepends the virtual environment’s directory for executables to the PATH environment variable for the running shell.

There should be no need in other circumstances to activate a virtual environment—scripts installed into virtual environments have a “shebang” line which points to the virtual environment’s Python interpreter. This means that the script will run with that interpreter regardless of the value of PATH. On Windows, “shebang” line processing is supported if you have the Python Launcher for Windows installed (this was added to Python in 3.3 - see PEP 397
for more details). Thus, double-clicking an installed script in a Windows Explorer window should run the script with the correct interpreter without there needing to be any reference to its virtual environment in PATH.
API

The high-level method described above makes use of a simple API which provides mechanisms for third-party virtual environment creators to customize environment creation according to their needs, the `EnvBuilder` class.

```python
class venv.EnvBuilder(system_site_packages=False, clear=False, symlinks=False, upgrade=False, with_pip=False, prompt=None)
```

The `EnvBuilder` class accepts the following keyword arguments on instantiation:

- `system_site_packages` – a Boolean value indicating that the system Python site-packages should be available to the environment (defaults to `False`).
- `clear` – a Boolean value which, if true, will delete the contents of any existing target directory, before creating the environment.
- `symlinks` – a Boolean value indicating whether to attempt to symlink the Python binary rather than copying.
- `upgrade` – a Boolean value which, if true, will upgrade an existing environment with the running Python - for use when that Python has been upgraded in-place (defaults to `False`).
- `with_pip` – a Boolean value which, if true, ensures pip is installed in the virtual environment. This uses `ensurepip` with the `--default-pip` option.
- `prompt` – a String to be used after virtual environment is activated (defaults to `None` which means directory name of the environment would be used).

*Changed in version 3.4: Added the `with_pip` parameter*
New in version 3.6: Added the prompt parameter

Creators of third-party virtual environment tools will be free to use the provided EnvBuilder class as a base class.

The returned env-builder is an object which has a method, create:

**create**(env_dir)

This method takes as required argument the path (absolute or relative to the current directory) of the target directory which is to contain the virtual environment. The `create` method will either create the environment in the specified directory, or raise an appropriate exception.

The `create` method of the EnvBuilder class illustrates the hooks available for subclass customization:

```python
def create(self, env_dir):
    """
    Create a virtualized Python environment in
    env_dir is the target directory to create a
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Each of the methods `ensure_directories()`, `create_configuration()`, `setup_python()`, `setup_scripts()` and `post_setup()` can be overridden.
**ensure_directories**(*env_dir*)

Creates the environment directory and all necessary directories, and returns a context object. This is just a holder for attributes (such as paths), for use by the other methods. The directories are allowed to exist already, as long as either `clear` or `upgrade` were specified to allow operating on an existing environment directory.

**create_configuration**(*context*)

Creates the pyvenv.cfg configuration file in the environment.

**setup_python**(*context*)

Creates a copy or symlink to the Python executable in the environment. On POSIX systems, if a specific executable `python3.x` was used, symlinks to `python` and `python3` will be created pointing to that executable, unless files with those names already exist.

**setup_scripts**(*context*)

Installs activation scripts appropriate to the platform into the virtual environment.

**post_setup**(*context*)

A placeholder method which can be overridden in third party implementations to pre-install packages in the virtual environment or perform other post-creation steps.

*Changed in version 3.7.2:* Windows now uses redirector scripts for `python[w].exe` instead of copying the actual binaries. In 3.7.2 only `setup_python()` does nothing unless running from a build in the source tree.
Changed in version 3.7.3: Windows copies the redirector scripts as part of `setup_python()` instead of `setup_scripts()`. This was not the case in 3.7.2. When using symlinks, the original executables will be linked.

In addition, `EnvBuilder` provides this utility method that can be called from `setup_scripts()` or `post_setup()` in subclasses to assist in installing custom scripts into the virtual environment.

```python
install_scripts(context, path)
```

`path` is the path to a directory that should contain subdirectories “common”, “posix”, “nt”, each containing scripts destined for the bin directory in the environment. The contents of “common” and the directory corresponding to `os.name` are copied after some text replacement of placeholders:

- `__VENV_DIR__` is replaced with the absolute path of the environment directory.
- `__VENV_NAME__` is replaced with the environment name (final path segment of environment directory).
- `__VENV_PROMPT__` is replaced with the prompt (the environment name surrounded by parentheses and with a following space)
- `__VENV_BIN_NAME__` is replaced with the name of the bin directory (either bin or Scripts).
- `__VENV_PYTHON__` is replaced with the absolute path of the environment's executable.

The directories are allowed to exist (for when an existing environment is being upgraded).

There is also a module-level convenience function:

```python
venv.create(env_dir, system_site_packages=False, clear=False,
```
Create an `EnvBuilder` with the given keyword arguments, and call its `create()` method with the `env_dir` argument.

*Changed in version 3.4: Added the `with_pip` parameter*
An example of extending EnvBuilder

The following script shows how to extend EnvBuilder by implementing a subclass which installs setuptools and pip into a created virtual environment:

```python
import os
import os.path
from subprocess import Popen, PIPE
import sys
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that
    easy_install other packages into the created virtual environment.
    
    :param nodist: If True, setuptools and pip are not installed into
        the created virtual environment.
    :param nopip: If True, pip is not installed into the created
        virtual environment.
    :param progress: If setuptools or pip are installed, the
        installation can be monitored by a progress callable. If specified,
        it is called with two arguments: a string indicating
        the progress, and a string indicating where the string
        is coming from. The context argument can have one
        of three values: 'main', indicating that it is called
        from virtualize itself, and 'stdout' and 'stderr'
        by reading lines from the output streams of the subprocess
        which is used to install the app.
    """
```
If a callable is not specified, information is output to sys.stderr.

```python
def __init__(self, *args, **kwargs):
    self.nodist = kwargs.pop('nodist', False)
    self.nopip = kwargs.pop('nopip', False)
    self.progress = kwargs.pop('progress', None)
    self.verbose = kwargs.pop('verbose', False)
    super().__init__(*args, **kwargs)

def post_setup(self, context):
    """
    Set up any packages which need to be pre-installed into the
    virtual environment being created.
    
    :param context: The information for the virtual environment
    creation request being processed.
    """
    os.environ['VIRTUAL_ENV'] = context.env_dir
    if not self.nodist:
        self.install_setuptools(context)
        # Can't install pip without setuptools
    if not self.nopip and not self.nodist:
        self.install_pip(context)

def reader(self, stream, context):
    """
    Read lines from a subprocess' output stream
    callable (if specified) or write progress information
    """
    progress = self.progress
    while True:
        s = stream.readline()
        if not s:
            break
        if progress is not None:
            ...
progress(s, context)
    if not self.verbose:
        sys.stderr.write('.')
    else:
        sys.stderr.write(s.decode('utf-8'))
        sys.stderr.flush()
stream.close()

def install_script(self, context, name, url):
    _, _, path, _, _, _ = urlparse(url)
    fn = os.path.split(path)[-1]
    binpath = context.bin_path
    distpath = os.path.join(binpath, fn)
    # Download script into the virtual environment
    urlretrieve(url, distpath)
    progress = self.progress
    if self.verbose:
        term = '\n'
    else:
        term = ''
    if progress is not None:
        progress('Installing %s ...%s' % (name, term))
    else:
        sys.stderr.write('Installing %s ...%s' % (name, term))
        sys.stderr.flush()
    # Install in the virtual environment
    args = [context.env_exe, fn]
    p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=context.bin_path)
    t1 = Thread(target=self.reader, args=(p.stdout,))
    t1.start()
    t2 = Thread(target=self.reader, args=(p.stderr,))
    t2.start()
    p.wait()
    t1.join()
    t2.join()
    if progress is not None:
progress('done.', 'main')
else:
    sys.stderr.write('done.
')
    # Clean up - no longer needed
    os.unlink(distpath)

def install_setuptools(self, context):
    """
    Install setuptools in the virtual environment.

    :param context: The information for the virtual environment creation request being processed.
    """
    url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
    self.install_script(context, 'setuptools', url)

    # clear up the setuptools archive which gets downloaded
    pred = lambda o: o.startswith('setuptools-')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:
        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    """
    Install pip in the virtual environment.

    :param context: The information for the virtual environment creation request being processed.
    """
    url = 'https://raw.github.com/pypa/pip/master/contrib/get-pip.py'
    self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
compatible = False

if not compatible:
    raise ValueError('This script is only for use with Python 3.3 or later')

else:
    import argparse

    parser = argparse.ArgumentParser(prog=__name__, description=

    parser.add_argument('dirs', metavar='ENV_DIR', help='A directory in which to create the virtual environment.')
    parser.add_argument('--no-setuptools', default=False, action='store_true', dest='no_setuptools', help='Don’t install setuptools in the virtual environment.')
    parser.add_argument('--no-pip', default=False, action='store_true', dest='no_pip', help='Don’t install pip in the virtual environment.')
    parser.add_argument('--system-site-packages', action='store_true', dest='system_site_packages', help='Give the virtual environment access to the system site-packages dir.

if os.name == 'nt':
    use_symlinks = False
else:
    use_symlinks = True

parser.add_argument('--symlinks', default=use_symlinks, action='store_true', dest='symlinks', help='Try to use symlinks rather than copies, when symlinks are the default for the platform.')

parser.add_argument('--clear', default=False, help='Clear out the virtual environment before creating it.')
dest='clear', help='Delete the contents of the virtual environment directory if it already exists, before virtual environment creation.'

parser.add_argument('--upgrade', default=False, dest='upgrade', help='Upgrade the virtual environment directory to use this version of Python, assuming Python has been upgraded in-place.'

parser.add_argument('--verbose', default=False, dest='verbose', help='Display the output from the scripts which install setuptools and pip.'

options = parser.parse_args(args)
if options.upgrade and options.clear:
    raise ValueError('you cannot supply --upgrade and --clear together.'

builder = ExtendedEnvBuilder(system_site_packages=clear=options
    symlinks=options
    upgrade=options
    nodist=options
    nopip=options
    verbose=options

for d in optionsdirs:
    builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
This script is also available for download online.
zipapp — Manage executable Python zip archives

New in version 3.5.

Source code: Lib/zipapp.py

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a Command-Line Interface and a Python API.
Basic Example

The following example shows how the Command-Line Interface can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the main function from the module myapp in the archive.

$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
Command-Line Interface

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If `source` is a directory, this will create an archive from the contents of `source`. If `source` is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the `–info` option is specified).

The following options are understood:

- `-o <output>, --output=<output>`
  Write the output to a file named `output`. If this option is not specified, the output filename will be the same as the input `source`, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

  An output filename must be specified if the `source` is an archive (and in that case, `output` must not be the same as `source`).

- `-p <interpreter>, --python=<interpreter>`
  Add a `#!` line to the archive specifying `interpreter` as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

- `-m <mainfn>, --main=<mainfn>`
  Write a `__main__.py` file to the archive that executes `mainfn`. The `mainfn` argument should have the form “pkg.mod:fn”, where “pkg.mod” is a package/module in the archive, and “fn” is a callable
in the given module. The __main__.py file will execute that callable.

--main cannot be specified when copying an archive.

-c, --compress
Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

--compress has no effect when copying an archive.

New in version 3.7.

--info
Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help
Print a short usage message and exit.
Python API

The module defines two convenience functions:

```python
zipapp.create_archive(source, target=None,
interpreted=None, main=None, filter=None, compressed=False)
```

Create an application archive from `source`. The source can be any of the following:

- The name of a directory, or a path-like object referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a path-like object referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the `interpreted` argument). The file name should include the `.pyz` extension, if required.
- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The `target` argument determines where the resulting archive will be written:

- If it is the name of a file, or a path-like object, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or None), the source must be a directory and the target will be a file with the same name as the source, with a .pyz extension added.

The `interpreted` argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a
“shebang” line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the interpreter results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The main argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a __main__.py file. The main argument should take the form “pkg.module:callable” and the archive will be run by importing “pkg.module” and executing the given callable with no arguments. It is an error to omit main if the source is a directory and does not contain a __main__.py file, as otherwise the resulting archive would not be executable.

The optional filter argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return True if the file is to be added.

The optional compressed argument determines whether files are compressed. If set to True, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

If a file object is specified for source or target, it is the caller’s responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need read and readline, or write methods. When creating an archive from a directory, if the target is a file object it will be passed to the zipfile.ZipFile class, and must supply the methods
needed by that class.

New in version 3.7: Added the filter and compressed arguments.

zipapp.get_interpreter(archive)
Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return None. The archive argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.
Examples

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the `create_archive()` function:

```
>>> import zipapp
>>> zipapp.create_archive('myapp.pyz', 'myapp')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the `create_archive()` function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
...     f.write(temp.getvalue())
Specifying the Interpreter

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX #! line, but there are other issues to consider:

- If you use “/usr/bin/env python” (or other forms of the “python” command, such as “/usr/bin/python”), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example “/usr/bin/env python3” your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say “python X.Y or later”, so be careful of using an exact version like “/usr/bin/env python3.4” as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an “/usr/bin/env python2” or “/usr/bin/env python3”, depending on whether your code is written for Python 2 or 3.
Creating Standalone Applications with zipapp

Using the `zipapp` module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application’s dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.

2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --tag=
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).

3. Optionally, delete the `.dist-info` directories created by pip in the `myapp` directory. These hold metadata for pip to manage the packages, and as you won't be making any further use of pip they aren't required - although it won't do any harm if you leave them.

4. Package the application using:
This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See Specifying the Interpreter for details. It can be shipped to users as a single file.

On Unix, the myapp.pyz file is executable as it stands. You can rename the file to remove the .pyz extension if you prefer a “plain” command name. On Windows, the myapp.pyz[w] file is executable by virtue of the fact that the Python interpreter registers the .pyz and .pyzw file extensions when installed.

Making a Windows executable

On Windows, registration of the .pyz extension is optional, and furthermore, there are certain places that don’t recognise registered extensions “transparently” (the simplest example is that subprocess.run(['myapp']) won’t find your application - you need to explicitly specify the extension).

On Windows, therefore, it is often preferable to create an executable from the zipapp. This is relatively easy, although it does require a C compiler. The basic approach relies on the fact that zipfiles can have arbitrary data prepended, and Windows exe files can have arbitrary data appended. So by creating a suitable launcher and tacking the .pyz file onto the end of it, you end up with a single-file executable that runs your application.

A suitable launcher can be as simple as the following:

```
#define Py_LIMITED_API 1
#include "Python.h"
```
If you define the WINDOWS preprocessor symbol, this will generate a GUI executable, and without it, a console executable.

To compile the executable, you can either just use the standard MSVC command line tools, or you can take advantage of the fact that distutils knows how to compile Python source:
The resulting launcher uses the “Limited ABI”, so it will run unchanged with any version of Python 3.x. All it needs is for Python (python3.dll) to be on the user’s PATH.

For a fully standalone distribution, you can distribute the launcher with your application appended, bundled with the Python “embedded” distribution. This will run on any PC with the appropriate architecture (32 bit or 64 bit).

Caveats

There are some limitations to the process of bundling your application into a single file. In most, if not all, cases they can be addressed without needing major changes to your application.

1. If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the
filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user’s machine).

2. If you are shipping a Windows executable as described above, you either need to ensure that your users have `python3.dll` on their PATH (which is not the default behaviour of the installer) or you should bundle your application with the embedded distribution.

3. The suggested launcher above uses the Python embedding API. This means that in your application, `sys.executable` will be your application, and *not* a conventional Python interpreter. Your code and its dependencies need to be prepared for this possibility. For example, if your application uses the `multiprocessing` module, it will need to call `multiprocessing.set_executable()` to let the module know where to find the standard Python interpreter.
The Python Zip Application Archive Format

Python has been able to execute zip files which contain a __main__.py file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a __main__.py file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on sys.path and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX “shebang” line to the file (#!/path/to/interpreter).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters b'#!' followed by an interpreter name, and then a newline (b'
') character. The interpreter name can be anything acceptable to the OS “shebang” processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in sys.getfilesystemencoding() on POSIX.

2. Standard zipfile data, as generated by the zipfile module. The zipfile content must include a file called __main__.py (which must be in the “root” of the zip file - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.
There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.
Python Runtime Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here’s an overview:

- **sys** — System-specific parameters and functions
- **sysconfig** — Provide access to Python’s configuration information
  - Configuration variables
  - Installation paths
  - Other functions
  - Using sysconfig as a script
- **builtins** — Built-in objects
- **__main__** — Top-level script environment
- **warnings** — Warning control
  - Warning Categories
  - The Warnings Filter
    - Describing Warning Filters
    - Default Warning Filter
    - Overriding the default filter
  - Temporarily Suppressing Warnings
  - Testing Warnings
  - Updating Code For New Versions of Dependencies
  - Available Functions
  - Available Context Managers
- **dataclasses** — Data Classes
  - Module-level decorators, classes, and functions
  - Post-init processing
  - Class variables
  - Init-only variables
  - Frozen instances
- Inheritance
- Default factory functions
- Mutable default values
- Exceptions

- **contextlib** — Utilities for with-statement contexts
  - Utilities
  - Examples and Recipes
    - Supporting a variable number of context managers
    - Catching exceptions from `__enter__` methods
    - Cleaning up in an `__enter__` implementation
    - Replacing any use of `try-finally` and flag variables
    - Using a context manager as a function decorator
  - Single use, reusable and reentrant context managers
    - Reentrant context managers
    - Reusable context managers

- **abc** — Abstract Base Classes
- **atexit** — Exit handlers
  - `atexit` Example
- **traceback** — Print or retrieve a stack traceback
  - `TracebackException` Objects
  - `StackSummary` Objects
  - `FrameSummary` Objects
  - `Traceback` Examples
- **`__future__`** — Future statement definitions
- **gc** — Garbage Collector interface
- **inspect** — Inspect live objects
  - Types and members
  - Retrieving source code
  - Introspecting callables with the `Signature` object
  - Classes and functions
  - The interpreter stack
  - Fetching attributes statically
  - Current State of Generators and Coroutines
- Code Objects Bit Flags
- Command Line Interface
- **site — Site-specific configuration hook**
- Readline configuration
- Module contents
**sys — System-specific parameters and functions**

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

**sys.abiflags**

On POSIX systems where Python was built with the standard configure script, this contains the ABI flags as specified by PEP 3149.

*New in version 3.2.*

**sys.argv**

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `-c`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the `fileinput` module.

**sys.base_exec_prefix**

Set during Python startup, before site.py is run, to the same value as `exec_prefix`. If not running in a virtual environment, the values will stay the same; if site.py finds that a virtual environment is in use, the values of `prefix` and `exec_prefix`
will be changed to point to the virtual environment, whereas `base_prefix` and `base_exec_prefix` will remain pointing to the base Python installation (the one which the virtual environment was created from).

*New in version 3.3.*

**sys.base_prefix**
Set during Python startup, before `site.py` is run, to the same value as `prefix`. If not running in a virtual environment, the values will stay the same; if `site.py` finds that a virtual environment is in use, the values of `prefix` and `exec_prefix` will be changed to point to the virtual environment, whereas `base_prefix` and `base_exec_prefix` will remain pointing to the base Python installation (the one which the virtual environment was created from).

*New in version 3.3.*

**sys.byteorder**
An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

**sys.builtin_module_names**
A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

**sys.call_tracing(func, args)**
Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from
a debugger from a checkpoint, to recursively debug some other code.

sys.**copyright**
A string containing the copyright pertaining to the Python interpreter.

sys.**_clear_type_cache()**
Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

sys.**_current_frames()**
Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the **traceback** module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

sys.**breakpointhook()**
This hook function is called by built-in **breakpoint()**. By default,
it drops you into the **pdb** debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. `pdb.set_trace()`) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in `breakpoint()` function passes its `*args` and `**kws` straight through. Whatever `breakpointhooks()` returns is returned from `breakpoint()`.

The default implementation first consults the environment variable `PYTHONBREAKPOINT`. If that is set to "0" then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, `pdb.set_trace()` is called. Otherwise this variable should name a function to run, using Python’s dotted-import nomenclature, e.g. `package.subpackage.module.function`. In this case, `package.subpackage.module` would be imported and the resulting module must have a callable named `function()`. This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in `breakpoint()` function.

Note that if anything goes wrong while importing the callable named by `PYTHONBREAKPOINT`, a `RuntimeWarning` is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, `PYTHONBREAKPOINT` is *not* consulted.

*New in version 3.7.*
sys._debugmallocstats()

Print low-level information to stderr about the state of CPython's memory allocator.

If Python is configured –with-pydebug, it also performs some expensive internal consistency checks.

*New in version 3.3.*

**CPython implementation detail:** This function is specific to CPython. The exact output format is not defined here, and may change.

sys.dllhandle

Integer specifying the handle of the Python DLL.

**Availability:** Windows.

sys.displayhook(value)

If `value` is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves `value` in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably 'strict'), encode it to `sys.stdout.encoding` with 'backslashreplace' error handler.

`sys.displayhook` is called on the result of evaluating an *expression* entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:
```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding,
                             if hasattr(sys.stdout, 'buffer'):
                                sys.stdout.buffer.write(bytes)
                             else:
                                text = bytes.decode(sys.stdout.encoding)
        sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

*Changed in version 3.2:* Use 'backslashreplace' error handler on *UnicodeEncodeError*.

**sys.dont_write_bytecode**

If this is true, Python won’t try to write *.pyc* files on the import of source modules. This value is initially set to *True* or *False* depending on the *-B* command line option and the *PYTHONDONOTWRITEBYTECODE* environment variable, but you can set it yourself to control bytecode file generation.

**sys.excepthook(type, value, traceback)**

This function prints out a given traceback and exception to *sys.stderr*.

When an exception is raised and uncaught, the interpreter calls
sys.excepthook with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to sys.excepthook.

sys.__breakpointhook__
sys.__displayhook__
sys.__excepthook__
These objects contain the original values of breakpointhook, displayhook, and excepthook at the start of the program. They are saved so that breakpointhook, displayhook and excepthook can be restored in case they happen to get replaced with broken or alternative objects.

*New in version 3.7:* __breakpointhook__

sys.exc_info()
This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, “handling an exception” is defined as “executing an except clause.” For any stack frame, only information about the exception being currently handled is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is:
**type** gets the type of the exception being handled (a subclass of `BaseException`); **value** gets the exception instance (an instance of the exception type); **traceback** gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

**sys.exec_prefix**
A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the **--exec-prefix** argument to the **configure** script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory `exec_prefix/lib/pythonX.Y/config`, and shared library modules are installed in `exec_prefix/lib/pythonX.Y/lib-dynload`, where `X.Y` is the version number of Python, for example 3.2.

**Note:** If a virtual environment is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via `base_exec_prefix`.

**sys.executable**
A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, `sys.executable` will be an empty string or `None`.

**sys.exit([arg])**
Exit from Python. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by finally clauses of `try`
statements are honored, and it is possible to intercept the exit attempt at an outer level.

The optional argument \texttt{arg} can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, \texttt{None} is equivalent to passing zero, and any other object is printed to \texttt{stderr} and results in an exit code of 1. In particular, \texttt{sys.exit("some error message")] is a quick way to exit a program when an error occurs.

Since \texttt{exit()} ultimately “only” raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

\textit{Changed in version 3.6}: If an error occurs in the cleanup after the Python interpreter has caught \texttt{SystemExit} (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

\texttt{sys.\texttt{flags}}

The \texttt{struct sequence} \texttt{flags} exposes the status of command line flags. The attributes are read only.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\texttt{attribute} & \texttt{flag} \\
\hline
\texttt{debug} & \texttt{-d} \\
\hline
\end{tabular}
\end{table}
<table>
<thead>
<tr>
<th>Option</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspect</td>
<td>-i</td>
</tr>
<tr>
<td>interactive</td>
<td>-i</td>
</tr>
<tr>
<td>isolated</td>
<td>-I</td>
</tr>
<tr>
<td>optimize</td>
<td>-0 or -00</td>
</tr>
<tr>
<td>dont_write_bytecode</td>
<td>-B</td>
</tr>
<tr>
<td>no_user_site</td>
<td>-s</td>
</tr>
<tr>
<td>no_site</td>
<td>-S</td>
</tr>
<tr>
<td>ignore_environment</td>
<td>-E</td>
</tr>
<tr>
<td>verbose</td>
<td>-v</td>
</tr>
<tr>
<td>bytes_warning</td>
<td>-b</td>
</tr>
<tr>
<td>quiet</td>
<td>-q</td>
</tr>
<tr>
<td>hash_randomization</td>
<td>-R</td>
</tr>
<tr>
<td>dev_mode</td>
<td>-X dev</td>
</tr>
<tr>
<td>utf8_mode</td>
<td>-X utf8</td>
</tr>
</tbody>
</table>

*Changed in version 3.2:* Added `quiet` attribute for the new `-q` flag.

*New in version 3.2.3:* The `hash_randomization` attribute.

*Changed in version 3.3:* Removed obsolete `division_warning` attribute.

*Changed in version 3.4:* Added `isolated` attribute for `-I isolated` flag.

*Changed in version 3.7:* Added `dev_mode` attribute for the new `-X dev` flag and `utf8_mode` attribute for the new `-X utf8` flag.
sys. **float_info**

A **struct sequence** holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

<table>
<thead>
<tr>
<th>attribute</th>
<th>float.h macro</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>DBL_EPSILON</td>
<td>difference between 1 and the least value greater than 1 that is representable as a float</td>
</tr>
<tr>
<td>dig</td>
<td>DBL_DIG</td>
<td>maximum number of decimal digits that can be faithfully represented in a float; see below</td>
</tr>
<tr>
<td>mant_dig</td>
<td>DBL_MANT_DIG</td>
<td>float precision: the number of base-radix digits in the significand of a float</td>
</tr>
<tr>
<td>max</td>
<td>DBL_MAX</td>
<td>maximum representable finite float</td>
</tr>
<tr>
<td>max_exp</td>
<td>DBL_MAX_EXP</td>
<td>maximum integer e such that <code>radix**(e-1)</code> is a representable finite float</td>
</tr>
<tr>
<td>max_10_exp</td>
<td>DBL_MAX_10_EXP</td>
<td>maximum integer e such that <code>10**e</code> is in the range of representable finite floats</td>
</tr>
<tr>
<td>min</td>
<td>DBL_MIN</td>
<td>minimum positive normalized float</td>
</tr>
<tr>
<td>min_exp</td>
<td>DBL_MIN_EXP</td>
<td>minimum integer e such that ( \text{radix}^{**(e-1)} ) is a normalized float</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>min_10_exp</td>
<td>DBL_MIN_10_EXP</td>
<td>minimum integer e such that ( 10^{**e} ) is a normalized float</td>
</tr>
<tr>
<td>radix</td>
<td>FLT_RADIX</td>
<td>radix of exponent representation</td>
</tr>
<tr>
<td>rounds</td>
<td>FLT_ROUNDS</td>
<td>integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.</td>
</tr>
</tbody>
</table>

The attribute `sys.float_info.dig` needs further explanation. If `s` is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value:

```python
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'  # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back again
'3.14159265358979'
```
But for strings with more than `sys.float_info.dig` significant digits, this isn’t always true:

```python
>>> s = '9876543211234567'  # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion change
'9876543211234568'
```

`sys.float_repr_style`
A string indicating how the `repr()` function behaves for floats. If the string has value `'short'` then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value `'legacy'` and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1.

*New in version 3.1.*

`sys.getallocatedblocks()`
Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter’s internal caches, the result can vary from call to call; you may have to call `_clear_type_cache()` and `gc.collect()` to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, `getallocatedblocks()` is allowed to return 0 instead.

*New in version 3.4.*
sys. `getandroidapilevel()`
Return the build time API version of Android as an integer.

Availability: Android.

New in version 3.7.

sys. `getcheckinterval()`
Return the interpreter’s “check interval”; see `setcheckinterval()`.

Deprecated since version 3.2: Use `getswitchinterval()` instead.

sys. `getdefaultencoding()`
Return the name of the current default string encoding used by the Unicode implementation.

sys. `getdlopenflags()`
Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

sys. `getfilesystemencoding()`
Return the name of the encoding used to convert between Unicode filenames and bytes filenames. For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system’s preferred representation.
This encoding is always ASCII-compatible.

`os.fsencode()` and `os.fsdecode()` should be used to ensure that the correct encoding and errors mode are used.

- In the UTF-8 mode, the encoding is utf-8 on any platform.
- On Mac OS X, the encoding is 'utf-8'.
- On Unix, the encoding is the locale encoding.
- On Windows, the encoding may be 'utf-8' or 'mbcs', depending on user configuration.

*Changed in version 3.2:* `getfilesystemencoding()` result cannot be `None` anymore.

*Changed in version 3.6:* Windows is no longer guaranteed to return 'mbcs'. See PEP 529 and `_enablelegacywindowsfsencoding()` for more information.

*Changed in version 3.7:* Return 'utf-8' in the UTF-8 mode.

**sys.getfilesystemencodeerrors()**

Return the name of the error mode used to convert between Unicode filenames and bytes filenames. The encoding name is returned from `getfilesystemencoding()`.

`os.fsencode()` and `os.fsdecode()` should be used to ensure that the correct encoding and errors mode are used.

*New in version 3.6.*

**sys.getrefcount**(object)

Return the reference count of the object. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.
**sys.getrecursionlimit()**
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by `setrecursionlimit()`.

**sys.getsizeof(object[, default])**
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, `default` will be returned if the object does not provide means to retrieve the size. Otherwise a `TypeError` will be raised.

`getsizeof()` calls the object's `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe for an example of using `getsizeof()` recursively to find the size of containers and all their contents.

**sys.getswitchinterval()**
Return the interpreter's "thread switch interval"; see `setswitchinterval()`.

*New in version 3.2.*

**sys._getframe([depth])**
Return a frame object from the call stack. If optional integer `depth`
is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, `ValueError` is raised. The default for `depth` is zero, returning the frame at the top of the call stack.

**CPython implementation detail:** This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

```python
sys.getprofile()
```
Get the profiler function as set by `setprofile()`.

```python
sys.gettrace()
```
Get the trace function as set by `settrace()`.

**CPython implementation detail:** The `gettrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

```python
sys.getwindowsversion()
```
Return a named tuple describing the Windows version currently running. The named elements are `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_minor`, `service_pack_major`, `suite_mask`, `product_type` and `platform_version`. `service_pack` contains a string, `platform_version` a 3-tuple and all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

`platform` will be 2 (VER_PLATFORM_WIN32_NT).
*product_type* may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (VER_NT_WORKSTATION)</td>
<td>The system is a workstation.</td>
</tr>
<tr>
<td>2 (VER_NT_DOMAIN_CONTROLLER)</td>
<td>The system is a domain controller.</td>
</tr>
<tr>
<td>3 (VER_NT_SERVER)</td>
<td>The system is a server, but not a domain controller.</td>
</tr>
</tbody>
</table>

This function wraps the Win32 GetVersionEx() function; see the Microsoft documentation on OSVERSIONINFOEX() for more information about these fields.

*platform_version* returns the accurate major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

**Availability:** Windows.

*Changed in version* 3.2: Changed to a named tuple and added
service_pack_minor, service_pack_major, suite_mask, and product_type.

*Changed in version* 3.6: Added *platform_version*

**sys.get_asyncgen_hooks()**

Returns an *asyncgen_hooks* object, which is similar to a
*namedtuple* of the form *(firstiter, finalizer)*, where *firstiter* and
*finalizer* are expected to be either *None* or functions which take an
asynchronous generator iterator as an argument, and are used to schedule finalization of an asynchronous generator by an event
loop.

*New in version 3.6:* See [PEP 525](https://www.python.org/dev/peps/pep-0525/) for more details.

**Note:** This function has been added on a provisional basis (see [PEP 411](https://www.python.org/dev/peps/pep-0411/) for details.)

`sys.get_coroutine_origin_tracking_depth()`  
Get the current coroutine origin tracking depth, as set by `set_coroutine_origin_tracking_depth()`.

*New in version 3.7.*

**Note:** This function has been added on a provisional basis (see [PEP 411](https://www.python.org/dev/peps/pep-0411/) for details.) Use it only for debugging purposes.

`sys.get_coroutine_wrapper()`  
Returns `None`, or a wrapper set by `set_coroutine_wrapper()`.

*New in version 3.5:* See [PEP 492](https://www.python.org/dev/peps/pep-0492/) for more details.

**Note:** This function has been added on a provisional basis (see [PEP 411](https://www.python.org/dev/peps/pep-0411/) for details.) Use it only for debugging purposes.

*Deprecated since version 3.7:* The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See [bpo-32591](https://bugs.python.org/issue32591) for details.

`sys.hash_info`  
A `struct sequence` giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see *Hashing of numeric types.*
<table>
<thead>
<tr>
<th>attribute</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>width in bits used for hash values</td>
</tr>
<tr>
<td>modulus</td>
<td>prime modulus P used for numeric hash scheme</td>
</tr>
<tr>
<td>inf</td>
<td>hash value returned for a positive infinity</td>
</tr>
<tr>
<td>nan</td>
<td>hash value returned for a nan</td>
</tr>
<tr>
<td>imag</td>
<td>multiplier used for the imaginary part of a complex number</td>
</tr>
<tr>
<td>algorithm</td>
<td>name of the algorithm for hashing of str, bytes, and memoryview</td>
</tr>
<tr>
<td>hash_bits</td>
<td>internal output size of the hash algorithm</td>
</tr>
<tr>
<td>seed_bits</td>
<td>size of the seed key of the hash algorithm</td>
</tr>
</tbody>
</table>

*New in version 3.2.*

*Changed in version 3.4: Added algorithm, hash_bits and seed_bits*

**sys.hexversion**

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```python
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful
when viewed as the result of passing it to the built-in `hex()` function. The struct sequence `sys.version_info` may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at API and ABI Versioning.

`sys.implementation`
An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

`name` is the implementation's identifier, e.g. 'cpython'. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

`version` is a named tuple, in the same format as `sys.version_info`. It represents the version of the Python implementation. This has a distinct meaning from the specific version of the Python language to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

`hexversion` is the implementation version in hexadecimal format, like `sys.hexversion`.

`cache_tag` is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the
implementation’s name and version, like 'cpython-33'. However, a Python implementation may use some other value if appropriate. If cache_tag is set to None, it indicates that module caching should be disabled.

```
sys.implementation
```
may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, `sys.implementation` will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See PEP 421 for more information.

New in version 3.3.

```
sys.int_info
```
A struct sequence that holds information about Python’s internal representation of integers. The attributes are read only.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits_per_digit</td>
<td>number of bits held in each digit. Python integers are stored internally in base $2^{\text{bits_per_digit}}$</td>
</tr>
<tr>
<td>sizeof_digit</td>
<td>size in bytes of the C type used to represent a digit</td>
</tr>
</tbody>
</table>

New in version 3.1.

```
sys.__interactivehook__
```
When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the PYTHONSTARTUP file is read, so that you can set this hook there. The site module sets this.
New in version 3.4.

sys.\texttt{intern(string)}

Enter \texttt{string} in the table of “interned” strings and return the interned string – which is \texttt{string} itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not immortal; you must keep a reference to the return value of \texttt{intern()} around to benefit from it.

sys.\texttt{is\_finalizing()}

Return \texttt{True} if the Python interpreter is shutting down, \texttt{False} otherwise.

New in version 3.5.

sys.\texttt{last\_type}
sys.\texttt{last\_value}
sys.\texttt{last\_traceback}

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is \texttt{import pdb; pdb.pm()} to enter the post-mortem debugger; see \texttt{pdb} module for more information.)
The meaning of the variables is the same as that of the return values from `exc_info()` above.

```
sys.maxsize
```
An integer giving the maximum value a variable of type `Py_ssize_t` can take. It's usually $2^{31} - 1$ on a 32-bit platform and $2^{63} - 1$ on a 64-bit platform.

```
sys.maxunicode
```
An integer giving the value of the largest Unicode code point, i.e. 1114111 (0x10FFFF in hexadecimal).

*Changed in version 3.3:* Before PEP 393, `sys.maxunicode` used to be either 0xFFFF or 0x10FFFF, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

```
sys.meta_path
```
A list of *meta path finder* objects that have their `find_spec()` methods called to see if one of the objects can find the module to be imported. The `find_spec()` method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a `module spec`, or `None` if the module cannot be found.

*See also:*
`importlib.abc.MetaPathFinder`
The abstract base class defining the interface of finder objects on `meta_path`.

`importlib.machinery.ModuleSpec`
The concrete class which `find_spec()` should return instances of.

*Changed in version 3.4:* Module specs were introduced in Python 3.4, by PEP 451. Earlier versions of Python looked for a method called `find_module()`. This is still called as a fallback if a `meta_path` entry doesn’t have a `find_spec()` method.

**sys.modules**

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail.

**sys.path**

A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of `PYTHONPATH`.

A program is free to modify this list for its own purposes. Only strings and bytes should be added to `sys.path`; all other data types are ignored during import.
See also: Module site This describes how to use .pth files to extend sys.path.

sys.\texttt{path\_hooks}

A list of callables that take a path argument to try to create a finder for the path. If a finder can be created, it is to be returned by the callable, else raise \texttt{ImportError}.

Originally specified in \texttt{PEP 302}.

sys.\texttt{path\_importer\_cache}

A dictionary acting as a cache for finder objects. The keys are paths that have been passed to \texttt{sys.path\_hooks} and the values are the finders that are found. If a path is a valid file system path but no finder is found on \texttt{sys.path\_hooks} then \texttt{None} is stored.

Originally specified in \texttt{PEP 302}.

\textit{Changed in version 3.3:} \texttt{None} is stored instead of \texttt{imp.NullImporter} when no finder is found.

sys.\texttt{platform}

This string contains a platform identifier that can be used to append platform-specific components to \texttt{sys.path}, for instance.

For Unix systems, except on Linux, this is the lowercased OS name as returned by \texttt{uname -s} with the first part of the version as returned by \texttt{uname -r} appended, e.g. 'sunos5' or 'freebsd8', \textit{at the time when Python was built}. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:
if sys.platform.startswith('freebsd '):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux '):
    # Linux-specific code here...

For other systems, the values are:

<table>
<thead>
<tr>
<th>System</th>
<th>platform value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>'linux'</td>
</tr>
<tr>
<td>Windows</td>
<td>'win32'</td>
</tr>
<tr>
<td>Windows/Cygwin</td>
<td>'cygwin'</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>'darwin'</td>
</tr>
</tbody>
</table>

*Changed in version 3.3:* On Linux, `sys.platform` doesn’t contain the major version anymore. It is always `linux`, instead of `linux2` or `linux3`. Since older Python versions include the version number, it is recommended to always use the `startswith` idiom presented above.

**See also:** `os.name` has a coarser granularity. `os.uname()` gives system-dependent version information. The `platform` module provides detailed checks for the system’s identity.

**sys.prefix**

A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string `'/usr/local'`. This can be set at build time with the `--prefix` argument to the `configure` script. The main collection of Python library modules is installed in the directory `prefix/lib/pythonX.Y` while the platform independent header
files (all except `pyconfig.h`) are stored in `prefix/include/pythonX.Y`, where `X.Y` is the version number of Python, for example 3.2.

**Note:** If a virtual environment is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`.

**sys.** `ps1`
**sys.** `ps2`
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'. . . '`. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

**sys.** `setcheckinterval(interval)`
Set the interpreter’s “check interval”. This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value $\leq 0$ checks every virtual instruction, maximizing responsiveness as well as overhead.

*Deprecated since version 3.2:* This function doesn’t have an effect anymore, as the internal logic for thread switching and asynchronous tasks has been rewritten. Use
setswitchinterval() instead.

sys.\texttt{setdlopenflags}(n)
Set the flags used by the interpreter for dlopen() calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as \texttt{sys.setdlopenflags(0)}. To share symbols across extension modules, call as \texttt{sys.setdlopenflags(os.RTLD_GLOBAL)}. Symbolic names for the flag values can be found in the \texttt{os} module (RTLD\_xxx constants, e.g. \texttt{os.RTLD\_LAZY}).

\textbf{Availability:} Unix.

sys.\texttt{setprofile}(profilefunc)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter \texttt{The Python Profilers} for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see \texttt{settrace()}), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return \texttt{None}. Error in the profile function will cause itself unset.

Profile functions should have three arguments: \texttt{frame}, \texttt{event}, and \texttt{arg}. \texttt{frame} is the current stack frame. \texttt{event} is a string: 'call', 'return', 'c\_call', 'c\_return', or 'c\_exception'. \texttt{arg} depends on the event type.
The events have the following meaning:

'call'
A function is called (or some other code block entered). The profile function is called; arg is None.

'return'
A function (or other code block) is about to return. The profile function is called; arg is the value that will be returned, or None if the event is caused by an exception being raised.

'c_call'
A C function is about to be called. This may be an extension function or a built-in. arg is the C function object.

'c_return'
A C function has returned. arg is the C function object.

'c_exception'
A C function has raised an exception. arg is the C function object.

def sys.setrecursionlimit(limit)
Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a RecursionError exception is raised.

Changed in version 3.5.1: A RecursionError exception is now
raised if the new limit is too low at the current recursion depth.

**sys.setswitchinterval**(interval)
Set the interpreter’s thread switch interval (in seconds). This floating-point value determines the ideal duration of the “timeslices” allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system’s decision. The interpreter doesn’t have its own scheduler.

*New in version 3.2.*

**sys.settrace**(tracefunc)
Set the system’s trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using `settrace()` for each thread being debugged.

Trace functions should have three arguments: `frame`, `event`, and `arg`. `frame` is the current stack frame. `event` is a string: 'call', 'line', 'return', 'exception' or 'opcode'. `arg` depends on the event type.

The trace function is invoked (with `event` set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or `None` if the scope shouldn’t be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.
If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

The events have the following meaning:

'call'
A function is called (or some other code block entered). The global trace function is called; `arg` is `None`; the return value specifies the local trace function.

'line'
The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; `arg` is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to `False` on that frame.

'return'
A function (or other code block) is about to return. The local trace function is called; `arg` is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception'
An exception has occurred. The local trace function is called; `arg` is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

'opcode'
The interpreter is about to execute a new opcode (see `dis` for opcode details). The local trace function is called; `arg` is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be
explicitly requested by setting `f_trace_opcodes` to `True` on the frame.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more information on code and frame objects, refer to The standard type hierarchy.

**CPython implementation detail:** The `settrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

*Changed in version 3.7:* 'opcode' event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

```python
sys.set_asyncgen_hooks(firstiter, finalizer)
```

Accepts two optional keyword arguments which are callables that accept an asynchronous generator iterator as an argument. The `firstiter` callable will be called when an asynchronous generator is iterated for the first time. The `finalizer` will be called when an asynchronous generator is about to be garbage collected.

*New in version 3.6:* See [PEP 525](https://www.python.org/dev/peps/pep-0525/) for more details, and for a reference example of a `finalizer` method see the implementation of `asyncio.Loop.shutdown_asyncgens` in `Lib/asyncio/base_events.py`

**Note:** This function has been added on a provisional basis (see [PEP 411](https://www.python.org/dev/peps/pep-0411/) for details.)
**sys.set_coroutine_origin_tracking_depth(depth)**

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be None.

To enable, pass a `depth` value greater than zero; this sets the number of frames whose information will be captured. To disable, pass set `depth` to zero.

This setting is thread-specific.

*New in version 3.7.*

**Note:** This function has been added on a provisional basis (see PEP 411 for details.) Use it only for debugging purposes.

**sys.set_coroutine_wrapper(wrapper)**

Allows intercepting creation of coroutine objects (only ones that are created by an async def function; generators decorated with `types.coroutine()` or `asyncio.coroutine()` will not be intercepted).

The `wrapper` argument must be either:

- a callable that accepts one argument (a coroutine object);
- `None`, to reset the wrapper.

If called twice, the new wrapper replaces the previous one. The function is thread-specific.

The `wrapper` callable cannot define new coroutines directly or indirectly:
```python
def wrapper(coro):
    async def wrap(coro):
        return await coro
        return wrap(coro)
    sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because `wrapper` creates a `wrap(coro)` coroutine:
foo()
```
See also `get_coroutine_wrapper()`.

*New in version 3.5:* See [PEP 492](https://www.python.org/dev/peps/pep-0492/) for more details.

**Note:** This function has been added on a provisional basis (see [PEP 411](https://www.python.org/dev/peps/pep-0411/) for details.) Use it only for debugging purposes.

*Deprecated since version 3.7:* The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See [bpo-32591](https://bugs.python.org/issue32591) for details.

```python
sys._enablelegacywindowsfsencoding()
```
Changes the default filesystem encoding and errors mode to `mbcs` and `replace` respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the `PYTHONLEGACYWINDOWSFSENCODING` environment variable before launching Python.

**Availability:** Windows.
New in version 3.6: See PEP 529 for more details.

```python
sys.stdin
sys.stdout
sys.stderr
```

**File objects** used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to `input()`);
- `stdout` is used for the output of `print()` and `expression` statements and for the prompts of `input()`;
- The interpreter’s own prompts and its error messages go to `stderr`.

These streams are regular **text files** like those returned by the `open()` function. Their parameters are chosen as follows:

- The character encoding is platform-dependent. Non-Windows platforms use the locale encoding (see `locale.getpreferredencoding()`).

On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns True) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system locale encoding if the process is not initially attached to a console.

The special behaviour of the console can be overridden by setting the environment variable `PYTHONLEGACYWINDOWSSTDIO` before starting Python. In
that case, the console codepages are used as for any other character device.

Under all platforms, you can override the character encoding by setting the `PYTHONIOENCODING` environment variable before starting Python or by using the new `-X utf8` command line option and `PYTHONUTF8` environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTDIO` is also set.

- When interactive, `stdout` and `stderr` streams are line-buffered. Otherwise, they are block-buffered like regular text files. You can override this value with the `-u` command-line option.

**Note:** To write or read binary data from/to the standard streams, use the underlying binary `buffer` object. For example, to write bytes to `stdout`, use `sys.stdout.buffer.write(b'abc')`. However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like `io.StringIO` which do not support the buffer attribute.

```python
sys.__stdin__
sys.__stdout__
sys.__stderr__
```

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.
It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

**Note:** Under some conditions stdin, stdout and stderr as well as the original values __stdin__, __stdout__ and __stderr__ can be None. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with `pythonw`.

**sys.thread_info**

A struct sequence holding information about the thread implementation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the thread implementation:</td>
</tr>
<tr>
<td></td>
<td>- 'nt': Windows threads</td>
</tr>
<tr>
<td></td>
<td>- 'pthread': POSIX threads</td>
</tr>
<tr>
<td></td>
<td>- 'solaris': Solaris threads</td>
</tr>
<tr>
<td>lock</td>
<td>Name of the lock implementation:</td>
</tr>
<tr>
<td></td>
<td>- 'semaphore': a lock uses a semaphore</td>
</tr>
<tr>
<td></td>
<td>- 'mutex+cond': a lock uses a mutex and a condition variable</td>
</tr>
<tr>
<td></td>
<td>- None if this information is unknown</td>
</tr>
<tr>
<td>version</td>
<td>Name and version of the thread library. It is a string, or None if this information is unknown.</td>
</tr>
</tbody>
</table>
New in version 3.3.

**sys.tracebacklimit**
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is 1000. When set to 0 or less, all traceback information is suppressed and only the exception type and value are printed.

**sys.version**
A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use `version_info` and the functions provided by the `platform` module.

**sys.api_version**
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

**sys.version_info**
A tuple containing the five components of the version number: `major`, `minor`, `micro`, `releaselevel`, and `serial`. All values except `releaselevel` are integers; the release level is 'alpha', 'beta', 'candidate', or 'final'. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.
Changed in version 3.1: Added named component attributes.

sys.\texttt{warnoptions}
This is an implementation detail of the warnings framework; do not modify this value. Refer to the \texttt{warnings} module for more information on the warnings framework.

sys.\texttt{winver}
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of \texttt{version}. It is provided in the \texttt{sys} module for informational purposes; modifying this value has no effect on the registry keys used by Python.

\textbf{Availability:} Windows.

sys.\texttt{_xoptions}
A dictionary of the various implementation-specific flags passed through the \texttt{-X} command-line option. Option names are either mapped to their values, if given explicitly, or to \texttt{True}. Example:

\begin{verbatim}
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license"
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
\end{verbatim}

\textbf{CPython implementation detail:} This is a CPython-specific way of accessing options passed through \texttt{-X}. Other implementations may export them through other means, or not at all.
New in version 3.2.

Citations


© Copyright 2001-2019, Python Software Foundation. The Python Software Foundation is a non-profit corporation. Please donate.

Last updated on Mar 25, 2019. Found a bug?

Created using Sphinx 1.8.5.
sysconfig — Provide access to Python’s configuration information

New in version 3.2.

Source code: Lib/sysconfig.py

The sysconfig module provides access to Python’s configuration information like the list of installation paths and the configuration variables relevant for the current platform.
Configuration variables

A Python distribution contains a Makefile and a pyconfig.h header file that are necessary to build both the Python binary itself and third-party C extensions compiled using distutils.

sysconfig puts all variables found in these files in a dictionary that can be accessed using get_config_vars() or get_config_var().

Notice that on Windows, it's a much smaller set.

sysconfig.get_config_vars(*args)

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return None.

sysconfig.get_config_var(name)

Return the value of a single variable name. Equivalent to get_config_vars().get(name).

If name is not found, return None.

Example of usage:

```python
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
```
/usr/local/lib

>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in `sysconfig` under unique identifiers based on the value returned by `os.name`.

Every new component that is installed using `distutils` or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- `posix_prefix`: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- `posix_home`: scheme for Posix platforms used when a `home` option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- `posix_user`: scheme for Posix platforms used when a component is installed through Distutils and the `user` option is used. This scheme defines paths located under the user home directory.
- `nt`: scheme for NT platforms like Windows.
- `nt_user`: scheme for NT platforms, when the `user` option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- `stdlib`: directory containing the standard Python library files that are not platform-specific.
- `platstdlib`: directory containing the standard Python library files that are platform-specific.
- **platlib**: directory for site-specific, platform-specific files.
- **purelib**: directory for site-specific, non-platform-specific files.
- **include**: directory for non-platform-specific header files.
- **platinCLUDE**: directory for platform-specific header files.
- **scripts**: directory for script files.
- **data**: directory for data files.

**sysconfig** provides some functions to determine these paths.

**sysconfig.** get_scheme_names()

Return a tuple containing all schemes currently supported in **sysconfig**.

**sysconfig.** get_path_names()

Return a tuple containing all path names currently supported in **sysconfig**.

**sysconfig.** get_path(name[, scheme[, vars[, expand]]])

Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

*name* has to be a value from the list returned by **get_path_names()**.

**sysconfig** stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

**get_path()** will use the variables returned by **get_config_vars()** to expand the path. All variables have default values for each platform so one may call this function and get the default value.
If `scheme` is provided, it must be a value from the list returned by `get_scheme_names()`. Otherwise, the default scheme for the current platform is used.

If `vars` is provided, it must be a dictionary of variables that will update the dictionary return by `get_config_vars()`.

If `expand` is set to False, the path will not be expanded using the variables.

If `name` is not found, return `None`.

```python
sysconfig.get_paths([scheme, vars, expand])
```

Return a dictionary containing all installation paths corresponding to an installation scheme. See `get_path()` for more information.

If `scheme` is not provided, will use the default scheme for the current platform.

If `vars` is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If `expand` is set to false, the paths will not be expanded.

If `scheme` is not an existing scheme, `get_paths()` will raise a `KeyError`.
Other functions

sysconfig.get_python_version()

Return the MAJOR.MINOR Python version number as a string. Similar to '%d.%d' % sys.version_info[:2].

sysconfig.get_platform()

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by ‘os.uname()’), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn’t particularly important.

Examples of returned values:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows will return one of:

- win-amd64 (64bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win32 (all others - specifically, sys.platform is returned)

Mac OS X can return:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat
For other non-POSIX platforms, currently just returns `sys.platform`.

`sysconfig.is_python_build()`  
Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

`sysconfig.parse_config_h(fp[, vars])`  
Parse a `config.h`-style file.  

`fp` is a file-like object pointing to the `config.h`-like file.  

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

`sysconfig.get_config_h_filename()`  
Return the path of `pyconfig.h`.

`sysconfig.get_makefile_filename()`  
Return the path of `Makefile`.  

Using `sysconfig` as a script

You can use `sysconfig` as a script with Python's `-m` option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
  data = "/usr/local"
  include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
  platinclude = "."
  platlib = "/usr/local/lib/python3.2/site-packages"
  platstdlib = "/usr/local/lib/python3.2"
  purelib = "/usr/local/lib/python3.2/site-packages"
  scripts = "/usr/local/bin"
  stdlib = "/usr/local/lib/python3.2"

Variables:
  AC_APPLE_UNIVERSAL_BUILD = "0"
  AIX_GENUINE_CPLUSPLUS = "0"
  AR = "ar"
  ARFLAGS = "rc"
  ...
```

This call will print in the standard output the information returned by `get_platform()`, `get_python_version()`, `get_path()` and `get_config_vars()`.
This module provides direct access to all ‘built-in’ identifiers of Python; for example, `builtins.open` is the full name for the built-in function `open()`. See Built-in Functions and Built-in Constants for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an `open()` function that wraps the built-in `open()`, this module can be used directly:

```python
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

As an implementation detail, most modules have the name...
`__builtins__` made available as part of their globals. The value of `__builtins__` is normally either this module or the value of this module's `__dict__` attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.
__main__ — Top-level script environment

'__main__' is the name of the scope in which top-level code executes. A module's __name__ is set equal to '__main__' when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own __name__, which allows a common idiom for conditionally executing code in a module when it is run as a script or with python -m but not when it is imported:

```python
if __name__ == '__main__':
    # execute only if run as a script
    main()
```

For a package, the same effect can be achieved by including a __main__.py module, the contents of which will be executed when the module is run with -m.
warnings — Warning control

Source code: Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_WarnEx()`; see Exception Handling for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling
resetwarnings().

The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

See also: `logging.captureWarnings()` allows you to handle all warnings with the standard logging infrastructure.
Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the `Warning` class.

The following warnings category classes are currently defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Warning</strong></td>
<td>This is the base class of all warning category classes. It is a subclass of <code>Exception</code>.</td>
</tr>
<tr>
<td><strong>UserWarning</strong></td>
<td>The default category for <code>warn()</code>.</td>
</tr>
<tr>
<td><strong>DeprecationWarning</strong></td>
<td>Base category for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in <code>__main__</code>).</td>
</tr>
<tr>
<td><strong>SyntaxWarning</strong></td>
<td>Base category for warnings about dubious syntactic features.</td>
</tr>
<tr>
<td>Warning Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>RuntimeWarning</code></td>
<td>about dubious runtime features.</td>
</tr>
<tr>
<td><code>FutureWarning</code></td>
<td>Base category for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.</td>
</tr>
<tr>
<td><code>PendingDeprecationWarning</code></td>
<td>Base category for warnings about features that will be deprecated in the future (ignored by default).</td>
</tr>
<tr>
<td><code>ImportWarning</code></td>
<td>Base category for warnings triggered during the process of importing a module (ignored by default).</td>
</tr>
<tr>
<td><code>UnicodeWarning</code></td>
<td>Base category for warnings related to Unicode.</td>
</tr>
<tr>
<td><code>BytesWarning</code></td>
<td>Base category for warnings related to <code>bytes</code> and <code>bytearray</code>.</td>
</tr>
<tr>
<td><code>ResourceWarning</code></td>
<td>Base category for warnings related to resource usage.</td>
</tr>
</tbody>
</table>

*Changed in version 3.7:* Previously `DeprecationWarning` and `FutureWarning` were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they're handled by the default warnings filters.
The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the filter determines the disposition of the match. Each entry is a tuple of the form \((\text{action}, \text{message}, \text{category}, \text{module}, \text{lineno})\), where:

- **action** is one of the following strings:
  - "default": print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
  - "error": turn matching warnings into exceptions
  - "ignore": never print matching warnings
  - "always": always print matching warnings
  - "module": print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)
  - "once": print only the first occurrence of matching warnings, regardless of location

- **message** is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.

- **category** is a class (a subclass of `Warning`) of which the warning
category must be a subclass in order to match.

- *module* is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.

- *lineno* is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the *Warning* class is derived from the built-in *Exception* class, to turn a warning into an error we simply raise `category(message)`.

If a warning is reported and doesn’t match any registered filter then the “default” action is applied (hence its name).

### Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in *The Warnings Filter*. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by
Commas, and the filters listed later take precedence over those listed before them (as they’re applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```plaintext
default           # Show all warnings (even those ignored by default
ignore           # Ignore all warnings
error            # Convert all warnings to errors
error::ResourceWarning # Treat ResourceWarning messages as errors
default::DeprecationWarning # Show DeprecationWarning messages
ignore,default::mymodule # Only report warnings triggered by "mymodule"
error::mymodule[.*] # Convert warnings to errors in "mymodule" and any subpackages of it
```

Default Warning Filter

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to `filterwarnings()`.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```plaintext
default::DeprecationWarning::__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In debug builds, the list of default warning filters is empty.
Changed in version 3.2: `DeprecationWarning` is now ignored by default in addition to `PendingDeprecationWarning`.

Changed in version 3.7: `DeprecationWarning` is once again shown by default when triggered directly by code in `__main__`.

Changed in version 3.7: `BytesWarning` no longer appears in the default filter list and is instead configured via `sys.warnoptions` when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide all Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The `sys.warnoptions` attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```python
import sys
if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that all warnings are displayed by default for the code under test, using code like:

```python
import sys
if not sys.warnoptions:
    import os, warnings
```
Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that `DeprecationWarning` messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```python
import warnings
warnings.filterwarnings("default", category=DeprecationWarning, module=user_ns.get)
```
Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the `catch_warnings` context manager:

```python
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the `catch_warnings` context manager at the same time, the behavior is undefined.
Testing Warnings

To test warnings raised by code, use the `catch_warnings` context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```python
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once/default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The `showwarning()` function in the
module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the `catch_warnings` context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).
Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this “ignored by default” list includes DeprecationWarning (for every module except __main__), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the unittest module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing -wd to the Python interpreter (this is shorthand for -W default) or setting PYTHONWARNINGS=default in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to -W (e.g. -W error). See the -W flag for more details on what is possible.
Available Functions

**warnings.warn**(message, category=None, stacklevel=1, source=None)

Issue a warning, or maybe ignore it or raise an exception. The category argument, if given, must be a warning category class (see above); it defaults to UserWarning. Alternatively message can be a Warning instance, in which case category will be ignored and message.__class__ will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The stacklevel argument can be used by wrapper functions written in Python, like this:

```python
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to deprecation()'s caller, rather than to the source of deprecation() itself (since the latter would defeat the purpose of the warning message).

source, if supplied, is the destroyed object which emitted a ResourceWarning.

*Changed in version 3.6:* Added source parameter.

**warnings.warn_explicit**(message, category, filename, lineno, module=None, registry=None, module_globals=None, source=None)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the
__warningregistry__ dictionary of the module). The module name defaults to the filename with .py stripped; if no registry is passed, the warning is never suppressed. message must be a string and category a subclass of Warning or message may be a Warning instance, in which case category will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a ResourceWarning.

*Changed in version 3.6:* Add the source parameter.

warnings.showwarning(message, category, filename, lineno, file=None, line=None)
Write a warning to a file. The default implementation calls formatwarning(message, category, filename, lineno, line) and writes the resulting string to file, which defaults to sys.stderr. You may replace this function with any callable by assigning to warnings.showwarning. line is a line of source code to be included in the warning message; if line is not supplied, showwarning() will try to read the line specified by filename and lineno.

warnings.formatwarning(message, category, filename, lineno, line=None)
Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. line is a line of source code to be included in the warning message; if line is
formatwarning() will try to read the line specified by filename and lineno.

```
warnings.filterwarnings(action, message=''
category=Warning, module=''
lineno=0, append=False)
```

Insert an entry into the list of warnings filter specifications. The entry is inserted at the front by default; if append is true, it is inserted at the end. This checks the types of the arguments, compiles the message and module regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

```
warnings.simplefilter(action, category=Warning, lineno=0
append=False)
```

Insert a simple entry into the list of warnings filter specifications. The meaning of the function parameters is as for filterwarnings(), but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

```
warnings.resetwarnings()
```

Reset the warnings filter. This discards the effect of all previous calls to filterwarnings(), including that of the -W command line options and calls to simplefilter().
Available Context Managers

```
class warnings.catch_warnings(*, record=False, module=None)
```

A context manager that copies and, upon exit, restores the warnings filter and the `showwarning()` function. If the `record` argument is `False` (the default) the context manager returns `None` on entry. If `record` is `True`, a list is returned that is progressively populated with objects as seen by a custom `showwarning()` function (which also suppresses output to `sys.stdout`). Each object in the list has attributes with the same names as the arguments to `showwarning()`.

The `module` argument takes a module that will be used instead of the module returned when you import `warnings` whose filter will be protected. This argument exists primarily for testing the `warnings` module itself.

**Note:** The `catch_warnings` manager works by replacing and then later restoring the module's `showwarning()` function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.
**dataclasses — Data Classes**

**Source code:** Lib/dataclasses.py

This module provides a decorator and functions for automatically adding generated special methods such as `_init__()` and `__repr__()` to user-defined classes. It was originally described in [PEP 557](https://www.python.org/dev/peps/pep-0557/).

The member variables to use in these generated methods are defined using [PEP 526](https://www.python.org/dev/peps/pep-0526/) type annotations. For example this code:

```python
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Will add, among other things, a `_init__()` that looks like:

```python
def __init__(self, name: str, unit_price: float, quantity_on_hand: int):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class: it is not directly specified in the `InventoryItem` definition shown above.
New in version 3.7.
Module-level decorators, classes, and functions

@dataclasses.dataclass(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)

This function is a decorator that is used to add generated special methods to classes, as described below.

The `dataclass()` decorator examines the class to find fields. A field is defined as class variable that has a type annotation. With two exceptions described below, nothing in `dataclass()` examines the type specified in the variable annotation.

The order of the fields in all of the generated methods is the order in which they appear in the class definition.

The `dataclass()` decorator will add various “dunder” methods to the class, described below. If any of the added methods already exist on the class, the behavior depends on the parameter, as documented below. The decorator returns the same class that is called on; no new class is created.

If `dataclass()` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `dataclass()` are equivalent:

```python
@dataclass
class C:
    ...
@dataclass()
```
class C:
...

@dataclass(init=True, repr=True, eq=True, order=False)
class C:
...

The parameters to \texttt{dataclass()} are:

- \texttt{init}: If true (the default), a \texttt{\_\_init\_()} method will be generated.
  
  If the class already defines \texttt{\_\_init\_()}, this parameter is ignored.

- \texttt{repr}: If true (the default), a \texttt{\_\_repr\_()} method will be generated. The generated \texttt{repr} string will have the class name and the name and \texttt{repr} of each field, in the order they are defined in the class. Fields that are marked as being excluded from the \texttt{repr} are not included. For example: \texttt{InventoryItem(name='widget', unit\_price=3.0, quantity\_on\_hand=10)}.

  If the class already defines \texttt{\_\_repr\_()}, this parameter is ignored.

- \texttt{eq}: If true (the default), an \texttt{\_\_eq\_()} method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

  If the class already defines \texttt{\_\_eq\_()}, this parameter is ignored.
- **order**: If true (the default is False), \_lt\_(), \_le\_(), \_gt\_(), and \_ge\_() methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If `order` is true and `eq` is false, a `ValueError` is raised.

If the class already defines any of \_lt\_(), \_le\_(), \_gt\_(), or \_ge\_(), then `TypeError` is raised.

- **unsafe_hash**: If False (the default), a \_hash\_() method is generated according to how `eq` and `frozen` are set.

\_hash\_() is used by built-in hash(), and when objects are added to hashed collections such as dictionaries and sets. Having a \_hash\_() implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer’s intent, the existence and behavior of \_eq\_(), and the values of the `eq` and `frozen` flags in the `dataclass()` decorator.

By default, `dataclass()` will not implicitly add a \_hash\_() method unless it is safe to do so. Neither will it add or change an existing explicitly defined \_hash\_() method. Setting the class attribute \_hash\_ = None has a specific meaning to Python, as described in the \_hash\_() documentation.

If \_hash\_() is not explicit defined, or if it is set to None, then `dataclass()` may add an implicit \_hash\_() method. Although not recommended, you can force `dataclass()` to create a \_hash\_() method with
unsafe_hash=True. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

Here are the rules governing implicit creation of a `__hash__()` method. Note that you cannot both have an explicit `__hash__()` method in your dataclass and set `unsafe_hash=True`; this will result in a `TypeError`.

If `eq` and `frozen` are both true, by default `dataclass()` will generate a `__hash__()` method for you. If `eq` is true and `frozen` is false, `__hash__()` will be set to None, marking it unhashable (which it is, since it is mutable). If `eq` is false, `__hash__()` will be left untouched meaning the `__hash__()` method of the superclass will be used (if the superclass is `object`, this means it will fall back to id-based hashing).

- `frozen`: If true (the default is False), assigning to fields will generate an exception. This emulates read-only frozen instances. If `__setattr__()` or `__delattr__()` is defined in the class, then `TypeError` is raised. See the discussion below.

Fields may optionally specify a default value, using normal Python syntax:

```python
@dataclass
class C:
    a: int  # 'a' has no default value
    b: int = 0  # assign a default value for 'b'
```
In this example, both `a` and `b` will be included in the added `__init__()` method, which will be defined as:

```python
def __init__(self, a: int, b: int = 0):
```

`TypeError` will be raised if a field without a default value follows a field with a default value. This is true either when this occurs in a single class, or as a result of class inheritance.

dataclasses.field(*, default=MISSING, default_factory=MISSING, repr=True, hash=None, init=True, compare=True, metadata=None)

For common and simple use cases, no other functionality is required. There are, however, some dataclass features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided `field()` function. For example:

```python
@dataclass
class C:
    mylist: List[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

As shown above, the `MISSING` value is a sentinel object used to detect if the `default` and `default_factory` parameters are provided. This sentinel is used because `None` is a valid value for `default`. No code should directly use the `MISSING` value.

The parameters to `field()` are:

- `default`: If provided, this will be the default value for this
field. This is needed because the `field()` call itself replaces the normal position of the default value.

- `default_factory`: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both `default` and `default_factory`.

- `init`: If true (the default), this field is included as a parameter to the generated `__init__()` method.

- `repr`: If true (the default), this field is included in the string returned by the generated `__repr__()` method.

- `compare`: If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).

- `hash`: This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If `None` (the default), use the value of `compare`: this would normally be the expected behavior. A field should be considered in the hash if it’s used for comparisons. Setting this value to anything other than `None` is discouraged.

One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type’s hash value. Even if a field is excluded from the hash, it will still be used for comparisons.

- `metadata`: This can be a mapping or `None`. `None` is treated
as an empty dict. This value is wrapped in MappingProxyType() to make it read-only, and exposed on the Field object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.

If the default value of a field is specified by a call to field(), then the class attribute for this field will be replaced by the specified default value. If no default is provided, then the class attribute will be deleted. The intent is that after the dataclass() decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```python
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute C.z will be 10, the class attribute C.t will be 20, and the class attributes C.x and C.y will not be set.

class dataclasses.Field

Field objects describe each defined field. These objects are created internally, and are returned by the fields() module-level method (see below). Users should never instantiate a Field object directly. Its documented attributes are:

- name: The name of the field.
- type: The type of the field.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, and `metadata` have the identical meaning and values as they do in the `field()` declaration.

Other attributes may exist, but they are private and must not be inspected or relied on.

**dataclasses.**`fields(class_or_instance)`

Returns a tuple of `Field` objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises `TypeError` if not passed a dataclass or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.

**dataclasses.**`asdict(instance, *, dict_factory=dict)`

Converts the dataclass instance to a dict (by using the factory function `dict_factory`). Each dataclass is converted to a dict of its fields, as name: value pairs. dataclasses, dicts, lists, and tuples are recursed into. For example:

```python
@dataclass
class Point:
    x: int
    y: int

@dataclass
class C:
    mylist: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]
```
Raises `TypeError` if `instance` is not a dataclass instance.

dataclasses.**astuple**( `instance`, *, `tuple_factory=tuple` )

Converts the dataclass instance to a tuple (by using the factory function `tuple_factory`). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into.

Continuing from the previous example:

```python
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

Raises `TypeError` if `instance` is not a dataclass instance.

dataclasses.**make_dataclass**( `cls_name`, `fields`, *, `bases=()`,
`namespace=None`, `init=True`, `repr=True`, `eq=True`, `order=False`,
`unsafe_hash=False`, `frozen=False` )

Creates a new dataclass with name `cls_name`, fields as defined in `fields`, base classes as given in `bases`, and initialized with a namespace as given in `namespace`. `fields` is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for type. The values of `init`, `repr`, `eq`, `order`, `unsafe_hash`, and `frozen` have the same meaning as they do in `dataclass()`.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the `dataclass()` function to convert that class to a dataclass. This function is provided as a convenience. For example:
C = make_dataclass('C',
    [('x', int),
     'y',
     ('z', int, field(default=5))],
    namespace={'add_one': lambda s
)

Is equivalent to:

@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1

dataclasses.replace(instance, **changes)

Creates a new object of the same type of instance, replacing fields with values from changes. If instance is not a Data Class, raises TypeError. If values in changes do not specify fields, raises TypeError.

The newly returned object is created by calling the __init__() method of the dataclass. This ensures that __post_init__(), if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to replace() so that they can be passed to __init__() and __post_init__().

It is an error for changes to contain any fields that are defined as having init=False. A ValueError will be raised in this case.
Be forewarned about how `init=False` fields work during a call to `replace()`. They are not copied from the source object, but rather are initialized in `__post_init__()`, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

`dataclasses.is_dataclass(class_or_instance)`

Returns True if its parameter is a dataclass or an instance of one, otherwise returns False.

If you need to know if a class is an instance of a dataclass (and not a dataclass itself), then add a further check for `not isinstance(obj, type)`:

```python
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```
Post-init processing

The generated `__init__()` code will call a method named `__post_init__()`, if `__post_init__()` is defined on the class. It will normally be called as `self.__post_init__()`. However, if any `InitVar` fields are defined, they will also be passed to `__post_init__()` in the order they were defined in the class. If no `__init__()` method is generated, then `__post_init__()` will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```python
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how `replace()` handles `init=False` fields.
Class variables

One of two places where `dataclass()` actually inspects the type of a field is to determine if a field is a class variable as defined in PEP 526. It does this by checking if the type of the field is `typing.ClassVar`. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level `fields()` function.
Init-only variables

The other place where `dataclass()` inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level `fields()` function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional `__post_init__()` method. They are not otherwise used by dataclasses.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```python
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, `fields()` will return `Field` objects for `i` and `j`, but not for `database`. 
Frozen instances

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the `dataclass()` decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a `FrozenInstanceError` when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__()` cannot use simple assignment to initialize fields, and must use `object.__setattr__()`. 

Inheritance

When the dataclass is being created by the `dataclass()` decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at `object`) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:

```python
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

The final list of fields is, in order, `x`, `y`, `z`. The final type of `x` is `int`, as specified in class `C`.

The generated `__init__()` method for `C` will look like:

```python
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```
Default factory functions

If a `field()` specifies a `default_factory`, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```python
global: list = field(default_factory=list)
```

If a field is excluded from `__init__() (using `init=False` and the field also specifies `default_factory`, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.
Mutable default values

Python stores default member variable values in class attributes. Consider this example, not using dataclasses:

```python
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class C share the same class variable x, as expected.

Using dataclasses, if this code was valid:

```python
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

it would generate code similar to:

```python
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
```
```python
def add(self, element):
    self.x += element

assert D().x is D().x
```

This has the same issue as the original example using class C. That is, two instances of class D that do not specify a value for x when creating a class instance will share the same copy of x. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, dataclasses will raise a TypeError if it detects a default parameter of type list, dict, or set. This is a partial solution, but it does protect against many common errors.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```python
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```
Exceptions

`exception` dataclasses. `FrozenInstanceError`

Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`.
contextlib — Utilities for with-statement contexts

**Source code:** Lib/contextlib.py

This module provides utilities for common tasks involving the `with` statement. For more information see also [Context Manager Types](#) and [With Statement Context Managers](#).
Utilities

Functions and classes provided:

```python
class contextlib(AbstractContextManager)
    An abstract base class for classes that implement `object.__enter__()` and `object.__exit__()`. A default implementation for `object.__enter__()` is provided which returns `self` while `object.__exit__()` is an abstract method which by default returns `None`. See also the definition of Context Manager Types.

    New in version 3.6.

class contextlib(AbstractAsyncContextManager)
    An abstract base class for classes that implement `object.__aenter__()` and `object.__aexit__()`. A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of Asynchronous Context Managers.

    New in version 3.7.

@contextlib.contextmanager
    This function is a decorator that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

    While many objects natively support use in with statements, sometimes a resource needs to be managed that isn’t a context
manager in its own right, and doesn’t implement a `close()` method for use with `contextlib.closing`.

An abstract example would be the following to ensure correct resource management:

```python
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource
...    # Resource is released at the end of this block
...    # even if code in the block raises an exception
```

The function being decorated must return a `generator`-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement’s `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the
generator must reraise that exception. Otherwise the generator context manager will indicate to the with statement that the exception has been handled, and execution will resume with the statement immediately following the with statement.

`contextmanager()` uses `ContextDecorator` so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise “one-shot” context managers created by `contextmanager()` to meet the requirement that context managers support multiple invocations in order to be used as decorators).

*Changed in version 3.2:* Use of `ContextDecorator`.

```python
@contextlib.asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
```
```python
yield conn
finally:
    await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

New in version 3.7.

`contextlib.closing(thing)`

Return a context manager that closes `thing` upon completion of the block. This is basically equivalent to:

```python
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```python
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.
contextlib.nullcontext(enter_result=None)

Return a context manager that returns `enter_result` from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```python
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

An example using `enter_result`:

```python
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

*New in version 3.7.*

contextlib.suppress(*exceptions)*

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a with statement and then resumes execution with the first statement following the end of the
with statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

For example:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is reentrant.

*New in version 3.4.*

```
contextlib.redirect_stdout(new_target)
```

Context manager for temporarily redirecting `sys.stdout` to
another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of `help()` normally is sent to `sys.stdout`. You can capture that output in a string by redirecting the output to an `io.StringIO` object:

```python
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

To send the output of `help()` to a file on disk, redirect the output to a regular file:

```python
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of `help()` to `sys.stderr`:

```python
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on `sys.stdout` means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is reentrant.
New in version 3.4.

```
contextlib.redirect_stderr(new_target)
```

Similar to `redirect_stdout()` but redirecting `sys.stderr` to another file or file-like object.

This context manager is reentrant.

New in version 3.5.

```
class contextlib.ContextDecorator
```

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by `contextmanager()`, so you get this functionality automatically.

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```
This change is just syntactic sugar for any construct of the following form:

```python
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```python
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False

Note: As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple with statements. If this is not the case, then the original construct with the explicit with statement inside the function should be used.

New in version 3.2.

class contextlib.ExitStack

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```python
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for # All opened files will automatically be closed # the with statement, even if attempts to open # in the list raise an exception]
```

Each instance maintains a stack of registered callbacks that are
called in reverse order when the instance is closed (either explicitly or implicitly at the end of a with statement). Note that callbacks are not invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their __init__ method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested with statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

New in version 3.3.

**enter_context(cm)**

Enters a new context manager and adds its __exit__() method to the callback stack. The return value is the result of the context manager's own __enter__() method.

These context managers may suppress exceptions just as they normally would if used directly as part of a with statement.

**push(exit)**

Adds a context manager's __exit__() method to the callback
As __enter__ is not invoked, this method can be used to cover part of an __enter__() implementation with a context manager's own __exit__() method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's __exit__() method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager __exit__() methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

**callback**(callback, *args, **kwds)

Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

**pop_all()**

Transfers the callback stack to a fresh ExitStack instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed.
(either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an “all or nothing” operation as follows:

```python
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously closed automatically. If all files are opened,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()

Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.AsyncExitStack

An asynchronous context manager, similar to `ExitStack`, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The close() method is not implemented, `aclose()` must be used instead.

`enter_async_context(cm)`

Similar to `enter_context()` but expects an asynchronous context manager.

`push_async_exit(exit)`
Similar to `push()` but expects either an asynchronous context manager or a coroutine function.

**push_async_callback**(*callback, *args, **kwds*)

Similar to `callback()` but expects a coroutine function.

**aclose()**

Similar to `close()` but properly handles awaitables.

Continuing the example for `asynccontextmanager()`:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context
                   for i in range(5)]
    # All opened connections will automatically be
    # closed when the async with statement ends,
    # even if attempts to open a connection later in
    # the list raise an exception.
```

*New in version 3.7.*
Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by `contextlib`.

Supporting a variable number of context managers

The primary use case for `ExitStack` is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```python
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)

# Perform operations that use the acquired resources
```

As shown, `ExitStack` also makes it quite easy to use `with` statements to manage arbitrary resources that don’t natively support the context management protocol.

Catching exceptions from `__enter__` methods

It is occasionally desirable to catch exceptions from an `__enter__`
method implementation, without inadvertently catching exceptions from the \texttt{with} statement body or the context manager's \texttt{\_exit\_} method. By using \texttt{ExitStack} the steps in the context management protocol can be separated slightly in order to allow this:

```python
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle \_enter\_ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with \texttt{try/except/finally} statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then \texttt{ExitStack} can make it easier to handle various situations that can't be handled directly in a \texttt{with} statement.

**Cleaning up in an \_enter\_ implementation**

As noted in the documentation of \texttt{ExitStack.push()}, this method can be useful in cleaning up an already allocated resource if later steps in the \texttt{\_enter\_()} implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:
from contextlib import contextmanager, AbstractContextManager

class ResourceManager(AbstractContextManager):
    def __init__(self, acquire_resource, release_resource):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
            self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception.
            # Accordingly, we want to keep the resource and pass it back to our caller.
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}".format(resource)
                raise RuntimeError(msg)
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic.
        self.release_resource()
variables

A pattern you will sometimes see is a try-finally statement with a flag variable to indicate whether or not the body of the finally clause should be executed. In its simplest form (that can’t already be handled just by using an except clause instead), it looks something like this:

```python
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any try statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a with statement, and then later decide to skip executing that callback:

```python
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up
front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```python
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn’t already neatly bundled into a standalone function, then it is still possible to use the decorator form of `ExitStack.callback()` to declare the resource cleanup in advance:

```python
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
        result = perform_operation()
        if result:
            stack.pop_all()
```
Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

**Using a context manager as a function decorator**

*ContextDecorator* makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```python
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```python
with track_entry_and_exit('widget loader '):
    print('Some time consuming activity goes here')
```
load_widget()

And also as a function decorator:

```python
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`.

See also:

**PEP 343 - The “with” statement**

The specification, background, and examples for the Python `with` statement.
Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they’re used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using `contextmanager()` are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```python
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...```
Reentrant context managers

More sophisticated context managers may be “reentrant”. These context managers can not only be used in multiple `with` statements, but may also be used inside a `with` statement that is already using the same context manager.

`threading.RLock` is an example of a reentrant context manager, as are `suppress()` and `redirect_stdout()`. Here's a very simple example of reentrant use:

```python
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```
Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is not the same thing as being thread safe. `redirect_stdout()`, for example, is definitely not thread safe, as it makes a global modification to the system state by binding `sys.stdout` to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are “reusable” context managers (or, to be completely explicit, “reusable, but not reentrant” context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing `with` statement.

`threading.Lock` is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use `threading.RLock` instead).

Another example of a reusable, but not reentrant, context manager is `ExitStack`, as it invokes all currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```python
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
```
Leaving first context
Callback: from first context
>>> with stack:
...   stack.callback(print, "Callback: from second context")
...   print("Leaving second context")

Leaving second context
Callback: from second context
>>> with stack:
...   stack.callback(print, "Callback: from outer context")
...   with stack:
...     stack.callback(print, "Callback: from inner context")
...     print("Leaving inner context")
...     print("Leaving outer context")

Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate ExitStack instances instead of reusing a single instance avoids that problem:

```python
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...   outer_stack.callback(print, "Callback: from outer context")
...   with ExitStack() as inner_stack:
...     inner_stack.callback(print, "Callback: from inner context")
...     print("Leaving inner context")
```
```python
... print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

The Python Software Foundation is a non-profit corporation. Please donate.

Last updated on Mar 25, 2019. Found a bug?

Created using Sphinx 1.8.5.
This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in PEP 3119; see the PEP for why this was added to Python. (See also PEP 3141 and the numbers module regarding a type hierarchy for numbers based on ABCs.)

The collections module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the collections.abc submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

This module provides the metaclass ABCMeta for defining ABCs and a helper class ABC to alternatively define ABCs through inheritance:

```python
class abc.ABC
    A helper class that has ABCMeta as its metaclass. With this class, an abstract base class can be created by simply deriving from ABC avoiding sometimes confusing metaclass usage, for example:
```

```python
from abc import ABC

class MyABC(ABC):
    pass
```

Note that the type of ABC is still ABCMeta, therefore inheriting from ABC requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also
define an abstract base class by passing the metaclass keyword and using `ABCMeta` directly, for example:

```python
from abc import ABCMeta
class MyABC(metaclass=ABCMeta):
    pass
```

*New in version 3.4.*

`class abc.ABCMeta`

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as “virtual subclasses” – these and their descendants will be considered subclasses of the registering ABC by the built-in `issubclass()` function, but the registering ABC won’t show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via `super()`). [1]

Classes created with a metaclass of `ABCMeta` have the following method:

`register(subclass)`

Register `subclass` as a “virtual subclass” of this ABC. For example:

```python
from abc import ABC
class MyABC(ABC):
    pass
```
MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance(), MyABC)

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

Changed in version 3.4: To detect calls to register(), you can use the get_cache_token() function.

You can also override this method in an abstract base class:

__subclasshook__(subclass)
( Must be defined as a class method.)

Check whether subclass is considered a subclass of this ABC. This means that you can customize the behavior of issubclass further without the need to call register() on every class you want to consider a subclass of the ABC. (This class method is called from the __subclasscheck__() method of the ABC.)

This method should return True, False or NotImplemented. If it returns True, the subclass is considered a subclass of this ABC. If it returns False, the subclass is not considered a subclass of this ABC, even if it would normally be one. If it returns NotImplemented, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None
    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B
            return True
        return NotImplemented

MyIterable.register(Foo)

The ABC `MyIterable` defines the standard iterable method, `__iter__()`, as an abstract method. The implementation given here can still be called from subclasses. The `get_iterator()` method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The `__subclasshook__()` class method defined here says that any class that has an `__iter__()` method in its `__dict__` (or in
that of one of its base classes, accessed via the __mro__ list) is considered a MyIterable too.

Finally, the last line makes Foo a virtual subclass of MyIterable, even though it does not define an __iter__() method (it uses the old-style iterable protocol, defined in terms of __len__() and __getitem__()). Note that this will not make get_iterator available as a method of Foo, so it is provided separately.

The abc module also provides the following decorator:

@abc.abstractmethod
A decorator indicating abstract methods.

Using this decorator requires that the class’s metaclass is ABCMeta or is derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal ‘super’ call mechanisms. abstractmethod() may be used to declare abstract methods for properties and descriptors.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The abstractmethod() only affects subclasses derived using regular inheritance; “virtual subclasses” registered with the ABC’s register() method are not affected.

When abstractmethod() is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples:

```python
class C(ABC):
```
@abstractmethod
def my_abstract_method(self, ...):
...
@classmethod
@abstractmethod
def my_abstract_classmethod(cls, ...):
...
@staticmethod
@abstractmethod
def my_abstract_staticmethod(...):
...

@property
@abstractmethod
def my_abstract_property(self):
...

@property.setter
@abstractmethod
def my_abstract_property(self, val):
...

@abstractmethod
def _get_x(self):
...

@abstractmethod
def _set_x(self, val):
...
x = property(_get_x, _set_x)

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using __isabstractmethod__. In general, this attribute should be True if any of the methods used to compose the descriptor are abstract. For example, Python’s built-in property does the equivalent of:
class Descriptor:
    ...
    @property
def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__') for f in (self._fget, self._fset, self._fdel))

Note: Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the super() mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

The abc module also supports the following legacy decorators:

@abc.abstractclassmethod
  New in version 3.2.
  
  Deprecated since version 3.3: It is now possible to use classmethod with abstractmethod(), making this decorator redundant.

A subclass of the built-in classmethod(), indicating an abstract classmethod. Otherwise it is similar to abstractmethod().

This special case is deprecated, as the classmethod() decorator is now correctly identified as abstract when applied to an abstract method:

class C(ABC):
    @classmethod
    @abstractmethod
def my_abstract_classmethod(cls, ...):
@abc.abstractstaticmethod

New in version 3.2.

Deprecated since version 3.3: It is now possible to use `staticmethod` with `abstractmethod()`, making this decorator redundant.

A subclass of the built-in `staticmethod()`, indicating an abstract staticmethod. Otherwise it is similar to `abstractmethod()`.

This special case is deprecated, as the `staticmethod()` decorator is now correctly identified as abstract when applied to an abstract method:

```python
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.abstractproperty

Deprecated since version 3.3: It is now possible to use `property`, `property.getter()`, `property.setter()` and `property.deleter()` with `abstractmethod()`, making this decorator redundant.

A subclass of the built-in `property()`, indicating an abstract property.

This special case is deprecated, as the `property()` decorator is now correctly identified as abstract when applied to an abstract
The above example defines a read-only property; you can also define a read-write abstract property by appropriately marking one or more of the underlying methods as abstract:

```python
class C(ABC):
    @property
    def my_abstract_property(self):
        ...
```

If only some components are abstract, only those components need to be updated to create a concrete property in a subclass:

```python
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

The `abc` module also provides the following functions:

```python
abc.get_cache_token()
```

Returns the current abstract base class cache token.
The token is an opaque object (that supports equality testing) identifying the current version of the abstract base class cache for virtual subclasses. The token changes with every call to `ABCMeta.register()` on any ABC.

*New in version 3.4.*

**Footnotes**

[1] C++ programmers should note that Python's virtual base class concept is not the same as C++'s.
The `atexit` module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. `atexit` runs these functions in the reverse order in which they were registered; if you register A, B, and C, at interpreter termination time they will be run in the order C, B, A.

**Note:** The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when `os._exit()` is called.

*Changed in version 3.7:* When used with C-API subinterpreters, registered functions are local to the interpreter they were registered in.

```python
atexit.register(func, *args, **kwargs)
```

Register `func` as a function to be executed at termination. Any optional arguments that are to be passed to `func` must be passed as arguments to `register()`. It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if `sys.exit()` is called or the main module's execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless `SystemExit` is raised) and the exception information is saved. After all exit handlers have had a
chance to run the last exception to be raised is re-raised.

This function returns \texttt{func}, which makes it possible to use it as a decorator.

\texttt{atexit.\texttt{unregister}(\texttt{func})}

Remove \texttt{func} from the list of functions to be run at interpreter shutdown. After calling \texttt{unregister()}, \texttt{func} is guaranteed not to be called when the interpreter shuts down, even if it was registered more than once. \texttt{unregister()} silently does nothing if \texttt{func} was not previously registered.

\textbf{See also:}

\textbf{Module \texttt{readline}}

Useful example of \texttt{atexit} to read and write \texttt{readline} history files.
atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter's updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```python
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)

Positional and keyword arguments may also be passed to register() to be passed along to the registered function when it is called:

```python
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')
```
Usage as a decorator:

```python
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

This only works with functions that can be called without arguments.
traceback — Print or retrieve a stack traceback

Source code: Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the sys.last_traceback variable and returned as the third item from sys.exc_info().

The module defines the following functions:

traceback.print_tb(tb, limit=None, file=None)

Print up to limit stack trace entries from traceback object tb (starting from the caller’s frame) if limit is positive. Otherwise, print the last abs(limit) entries. If limit is omitted or None, all entries are printed. If file is omitted or None, the output goes to sys.stdout; otherwise it should be an open file or file-like object to receive the output.

Changed in version 3.5: Added negative limit support.

traceback.print_exception(etype, value, tb, limit=None, file=None, chain=True)

Print exception information and stack trace entries from traceback
object \(tb\) to \(file\). This differs from \texttt{print\_tb()} in the following ways:

- if \(tb\) is not \texttt{None}, it prints a header \texttt{Traceback} (most recent call last):
- it prints the exception \texttt{etype} and \texttt{value} after the stack trace
- if \texttt{type(value)} is \texttt{SyntaxError} and \texttt{value} has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional \texttt{limit} argument has the same meaning as for \texttt{print\_tb()}. If \texttt{chain} is true (the default), then chained exceptions (the \texttt{__cause__} or \texttt{__context__} attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

\textit{Changed in version 3.5:} The \texttt{etype} argument is ignored and inferred from the type of \texttt{value}.

\begin{verbatim}
\texttt{traceback.print\_exc(limit=None, file=None, chain=True)}
\end{verbatim}

This is a shorthand for \texttt{print\_exception(*sys.exc_info(), limit, file, chain)}.

\begin{verbatim}
\texttt{traceback.print\_last(limit=None, file=None, chain=True)}
\end{verbatim}

This is a shorthand for \texttt{print\_exception(sys.last\_type, sys.last\_value, sys.last\_traceback, limit, file, chain)}. In general it will work only after an exception has reached an interactive prompt (see \texttt{sys.last\_type}).

\begin{verbatim}
\texttt{traceback.print\_stack(f=None, limit=None, file=None)}
\end{verbatim}

Print up to \texttt{limit} stack trace entries (starting from the invocation point) if \texttt{limit} is positive. Otherwise, print the last \texttt{abs(limit)} entries. If \texttt{limit} is omitted or \texttt{None}, all entries are printed. The
optional $f$ argument can be used to specify an alternate stack frame to start. The optional $file$ argument has the same meaning as for $\text{print\_tb()}$.

*Changed in version 3.5:* Added negative $limit$ support.

**traceback.$\text{extract\_tb}(tb, limit=None)$**
Return a $\text{StackSummary}$ object representing a list of “pre-processed” stack trace entries extracted from the traceback object $tb$. It is useful for alternate formatting of stack traces. The optional $limit$ argument has the same meaning as for $\text{print\_tb()}$. A “pre-processed” stack trace entry is a $\text{FrameSummary}$ object containing attributes $\text{filename}$, $\text{lineno}$, $\text{name}$, and $\text{line}$ representing the information that is usually printed for a stack trace. The line is a string with leading and trailing whitespace stripped; if the source is not available it is None.

**traceback.$\text{extract\_stack}(f=None, limit=None)$**
Extract the raw traceback from the current stack frame. The return value has the same format as for $\text{extract\_tb()}$. The optional $f$ and $limit$ arguments have the same meaning as for $\text{print\_stack()}$.

**traceback.$\text{format\_list}(extracted\_list)$**
Given a list of tuples or $\text{FrameSummary}$ objects as returned by $\text{extract\_tb()}$ or $\text{extract\_stack()}$, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

**traceback.$\text{format\_exception\_only}(etype, value)$**
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

```
traceback.format_exception(etype, value, tb, limit=None, chain=True)
```

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

*Changed in version 3.5:* The `etype` argument is ignored and inferred from the type of `value`.

```
traceback.format_exc(limit=None, chain=True)
```

This is like `print_exc(limit)` but returns a string instead of printing to a file.

```
traceback.format_tb(tb, limit=None)
```

A shorthand for `format_list(extract_tb(tb, limit))`.

```
traceback.format_stack(f=None, limit=None)
```

A shorthand for `format_list(extract_stack(f, limit))`.

```
traceback.clear_frames(tb)
```
Clears the local variables of all the stack frames in a traceback `tb` by calling the `clear()` method of each frame object.

*New in version 3.4.*

**traceback.walk_stack(f)**

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If `f` is `None`, the current stack is used. This helper is used with `StackSummary.extract()`.

*New in version 3.5.*

**traceback.walk_tb(tb)**

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with `StackSummary.extract()`.

*New in version 3.5.*

The module also defines the following classes:
**TracebackException** Objects

*New in version 3.5.*

TracebackException objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

```python
class traceback.TracebackException(exc_type, exc_value, exc_traceback, *, limit=None, lookup_lines=True, capture_locals=False):
    Capture an exception for later rendering. limit, lookup_lines and capture_locals are as for the StackSummary class.
```

Note that when locals are captured, they are also shown in the traceback.

__cause__
A TracebackException of the original __cause__.

__context__
A TracebackException of the original __context__.

__suppress_context__
The __suppress_context__ value from the original exception.

stack
A StackSummary representing the traceback.

exc_type
The class of the original traceback.

filename
For syntax errors - the file name where the error occurred.

**lineno**
For syntax errors - the line number where the error occurred.

**text**
For syntax errors - the text where the error occurred.

**offset**
For syntax errors - the offset into the text where the error occurred.

**msg**
For syntax errors - the compiler error message.

```python
classmethod from_exception(exc, *, limit=None, lookup_lines=True, capture_locals=False)
```
Capture an exception for later rendering. *limit*, *lookup_lines* and
*capture_locals* are as for the StackSummary class.

Note that when locals are captured, they are also shown in the traceback.

```python
format(*, chain=True)
```
Format the exception.

If *chain* is not True, __cause__ and __context__ will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines.

**print_exception()** is a wrapper around this method which just prints the lines to a file.
The message indicating which exception occurred is always the last string in the output.

**format_exception_only()**  
Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for `SyntaxError` exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.
**StackSummary Objects**

New in version 3.5.

StackSummary objects represent a call stack ready for formatting.

```python
class traceback.StackSummary
    classmethod extract(frame_gen, *, limit=None, lookup_lines=True, capture_locals=False)
        Construct a StackSummary object from a frame generator (such as is returned by walk_stack() or walk_tb()).

        If limit is supplied, only this many frames are taken from frame_gen. If lookup_lines is False, the returned FrameSummary objects will not have read their lines in yet, making the cost of creating the StackSummary cheaper (which may be valuable if it may not actually get formatted). If capture_locals is True the local variables in each FrameSummary are captured as object representations.

    classmethod from_list(a_list)
        Construct a StackSummary object from a supplied list of FrameSummary objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

    format()
        Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.
```
For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

*Changed in version 3.6:* Long sequences of repeated frames are now abbreviated.
**FrameSummary** Objects

*New in version 3.5.*

**FrameSummary** objects represent a single frame in a traceback.

```python
class traceback.FrameSummary(filename, lineno, name, lookup_line=True, locals=None, line=None)
```

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If `lookup_line` is `False`, the source code is not looked up until the **FrameSummary** has the line attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. `locals` is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.
Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the code module.

```python
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```python
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]
```
```
try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
print("*** print_tb:")
traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
print("*** print_exception:"
   ="# exc_type below is ignored on 3.5 and later"
traceback.print_exception(exc_type, exc_value, exc_traceback, limit=2, file=sys.stdout)
print("*** print_exc:"
   ="# exc_type below is ignored on 3.5 and later"
traceback.print_exception(limit=2, file=sys.stdout)
print("*** format_exc, first and last line:")
formatted_lines = traceback.format_exc().splitlines()
print(formatted_lines[0])
print(formatted_lines[-1])
print("*** format_exception:"
   ="# exc_type below is ignored on 3.5 and later"
print(repr(traceback.format_exception(exc_type, exc_traceback, exc_traceback.tb_lineno)))
print("*** extract_tb:")
print(repr(traceback.extract_tb(exc_traceback)))
print("*** format_tb:"
print(repr(traceback.format_tb(exc_traceback))))
print("*** tb_lineno:", exc_traceback.tb_lineno)
```

The output for the example would look similar to this:

```plaintext
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
```
The following example shows the different ways to print and format the stack:

```python
>>> import traceback
>>> def another_function():
...     lumberjack()
... 
>>> def lumberjack():
```
...    traceback.print_stack()
...    print(repr(traceback.extract_stack()))
...    print(repr(traceback.format_stack()))
...

>>> another_function()
File "<doctest>", line 10, in <module>
    another_function()
File "<doctest>", line 3, in another_function
    lumberstack()
File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('doctest', 10, '<module>', 'another_function()')
 ('doctest', 3, 'another_function', 'lumberstack()')
 ('doctest', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')
]

>>> Another function
    File "<doctest>", line 10, in <module>
    another_function()
    File "<doctest>", line 3, in another_function
    lumberstack()
    File "<doctest>", line 7, in lumberstack
    traceback.print_stack()
[('doctest', 10, '<module>', 'another_function()')
 ('doctest', 3, 'another_function', 'lumberstack()')
 ('doctest', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')
]

This last example demonstrates the final few formatting functions:

>>> import traceback

>>> traceback.format_list([('spam.py', 3, '<module>'), ('eggs.py', 42, 'eggs',)
...
]

[ 'File "spam.py", line 3, in <module>
  spam.eggs()
',
 'File "eggs.py", line 42, in eggs
  return "bacon"
'
]

>>> an_error = IndexError('tuple index out of range'

>>> traceback.format_exception_only(type(an_error),
['IndexError: tuple index out of range
']

...
__future__ — Future statement definitions

Source code: Lib/__future__.py

__future__ is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions (the import of __future__ will fail, because there was no module of that name prior to 2.1).
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing __future__ and examining its contents.

Each statement in __future__.py is of the form:

```python
FeatureName = _Feature(OptionalRelease, MandatoryRelease, CompilerFlag)
```

where, normally, OptionalRelease is less than MandatoryRelease, and both are 5-tuples of the same form as sys.version_info:

```python
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or
 PY_RELEASE_SERIAL  # the 3; an int
```
OptionalRelease records the first release in which the feature was accepted.

In the case of a MandatoryRelease that has not yet occurred, MandatoryRelease predicts the release in which the feature will become part of the language.

Else MandatoryRelease records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be None, meaning that a planned feature got dropped.

Instances of class _Feature have two corresponding methods, getOptionalRelease() and getMandatoryRelease().

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function compile() to enable the feature in dynamically compiled code. This flag is stored in the compiler_flag attribute on _Feature instances.

No feature description will ever be deleted from __future__. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

<table>
<thead>
<tr>
<th>feature</th>
<th>optional in</th>
<th>mandatory in</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested_scopes</td>
<td>2.1.0b1</td>
<td>2.2</td>
<td>PEP 227: Statically Nested Scopes</td>
</tr>
<tr>
<td>Feature</td>
<td>Release</td>
<td>Python</td>
<td>PEP</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>generators</td>
<td>2.2.0a1</td>
<td>2.3</td>
<td><strong>PEP 255</strong>: Simple Generators</td>
</tr>
<tr>
<td>division</td>
<td>2.2.0a2</td>
<td>3.0</td>
<td><strong>PEP 238</strong>: Changing the Division Operator</td>
</tr>
<tr>
<td>absolute_import</td>
<td>2.5.0a1</td>
<td>3.0</td>
<td><strong>PEP 328</strong>: Imports: Multi-Line and Absolute/Relative</td>
</tr>
<tr>
<td>with_statement</td>
<td>2.5.0a1</td>
<td>2.6</td>
<td><strong>PEP 343</strong>: The “with” Statement</td>
</tr>
<tr>
<td>print_function</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td><strong>PEP 3105</strong>: Make print a function</td>
</tr>
<tr>
<td>unicode_literals</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td><strong>PEP 3112</strong>: Bytes literals in Python 3000</td>
</tr>
<tr>
<td>generator_stop</td>
<td>3.5.0b1</td>
<td>3.7</td>
<td><strong>PEP 479</strong>: StopIteration handling inside generators</td>
</tr>
<tr>
<td>annotations</td>
<td>3.7.0b1</td>
<td>4.0</td>
<td><strong>PEP 563</strong>: Postponed evaluation of annotations</td>
</tr>
</tbody>
</table>

See also:

**Future statements**
How the compiler treats future imports.
This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling \texttt{gc.disable()}. To debug a leaking program call \texttt{gc.set_debug(gc.DEBUG_LEAK)}. Notice that this includes \texttt{gc.DEBUG_SAVEALL}, causing garbage-collected objects to be saved in \texttt{gc.garbage} for inspection.

The \texttt{gc} module provides the following functions:

\texttt{gc.enable()}

Enable automatic garbage collection.

\texttt{gc.disable()}

Disable automatic garbage collection.

\texttt{gc.isenabled()}

Returns true if automatic collection is enabled.

\texttt{gc.collect\,(\textit{generation}=2)}

With no arguments, run a full collection. The optional argument \textit{generation} may be an integer specifying which generation to collect (from 0 to 2). A \texttt{ValueError} is raised if the generation number is invalid. The number of unreachable objects found is returned.
The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular float.

```python
gc.set_debug(flags)
```
Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

```python
gc.get_debug()
```
Return the debugging flags currently set.

```python
gc.get_objects()
```
Returns a list of all objects tracked by the collector, excluding the list returned.

```python
gc.get_stats()
```
Return a list of three per-generation dictionaries containing collection statistics since interpreter start. The number of keys may change in the future, but currently each dictionary will contain the following items:

- `collections` is the number of times this generation was collected;
- `collected` is the total number of objects collected inside this generation;
- `uncollectable` is the total number of objects which were found to be uncollectable (and were therefore moved to the garbage list) inside this generation.

*New in version 3.4.*
**gc.set_threshold(threshold0[, threshold1[, threshold2]])**

Set the garbage collection thresholds (the collection frequency). Setting `threshold0` to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds `threshold0`, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than `threshold1` times since generation 1 has been examined, then generation 1 is examined as well. Similarly, `threshold2` controls the number of collections of generation 1 before collecting generation 2.

**gc.get_count()**

Return the current collection counts as a tuple of `(count0, count1, count2)`.

**gc.get_threshold()**

Return the current collection thresholds as a tuple of `(threshold0, threshold1, threshold2)`.

**gc.get_referrers(*objs)**

Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.
Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call \texttt{collect()} before calling \texttt{get_referrers()}.

Care must be taken when using objects returned by \texttt{get_referrers()} because some of them could still be under construction and hence in a temporarily invalid state. Avoid using \texttt{get_referrers()} for any purpose other than debugging.

\texttt{gc.get_referents(*objs)}

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments’ C-level \texttt{tp_traverse} methods (if any), and may not be all objects actually directly reachable. \texttt{tp_traverse} methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

\texttt{gc.is_tracked(obj)}

Returns \texttt{True} if the object is currently tracked by the garbage collector, \texttt{False} otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```
>>> gc.is_tracked(0)
False
```
```python
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked("a": 1)
False
>>> gc.is_tracked("a": [])
True
```

New in version 3.1.

**gc**. **freeze()**

Freeze all the objects tracked by gc - move them to a permanent generation and ignore all the future collections. This can be used before a POSIX fork() call to make the gc copy-on-write friendly or to speed up collection. Also collection before a POSIX fork() call may free pages for future allocation which can cause copy-on-write too so it’s advised to disable gc in master process and freeze before fork and enable gc in child process.

New in version 3.7.

**gc**. **unfreeze()**

Unfreeze the objects in the permanent generation, put them back into the oldest generation.

New in version 3.7.

**gc**. **get_freeze_count()**

Return the number of objects in the permanent generation.

New in version 3.7.
The following variables are provided for read-only access (you can mutate the values but should not rebind them):

**gc.garbage**

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-NULL `tp_de1` slot.

If `DEBUG_SAVEALL` is set, then all unreachable objects will be added to this list rather than freed.

*Changed in version 3.2:* If this list is non-empty at `interpreter shutdown`, a `ResourceWarning` is emitted, which is silent by default. If `DEBUG_UNCOLLECTABLE` is set, in addition alluncollectable objects are printed.

*Changed in version 3.4:* Following [PEP 442](https://www.python.org/dev/peps/pep-0442/), objects with a `__del__() method don’t end up in `gc.garbage` anymore.

**gc.callbacks**

A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, `phase` and `info`.

`phase` can be one of two values:

“start”: The garbage collection is about to start.

“stop”: The garbage collection has finished.

`info` is a dict providing more information for the callback. The following keys are currently defined:
“generation”: The oldest generation being collected.

“collected”: When phase is “stop”, the number of objects successfully collected.

“uncollectable”: When phase is “stop”, the number of objects that could not be collected and were put in garbage.

Applications can add their own callbacks to this list. The primary use cases are:

Gathering statistics about garbage collection, such as how often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in garbage.

*New in version 3.3.*

The following constants are provided for use with `set_debug()`:

**gc**. **DEBUG_STATS**
Print statistics during collection. This information can be useful when tuning the collection frequency.

**gc**. **DEBUG_COLLECTABLE**
Print information on collectable objects found.

**gc**. **DEBUG_UNCOLLECTABLE**
Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the garbage list.
Changed in version 3.2: Also print the contents of the garbage list at interpreter shutdown, if it isn’t empty.

gc . DEBUG_SAVEALL

When set, all unreachable objects found will be appended to garbage rather than being freed. This can be useful for debugging a leaking program.

**gc . DEBUG_LEAK**

The debugging flags necessary for the collector to print information about a leaking program (equal to DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL).
The `inspect` module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.
Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The functions whose names begin with “is” are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>file</strong></td>
<td>filename (missing for built-in modules)</td>
</tr>
<tr>
<td>class</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this class was defined</td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong></td>
<td>qualified name</td>
</tr>
<tr>
<td></td>
<td><strong>module</strong></td>
<td>name of module in which this class was defined</td>
</tr>
<tr>
<td>method</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this method was defined</td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong></td>
<td>qualified name</td>
</tr>
<tr>
<td></td>
<td><strong>func</strong></td>
<td>function object containing implementation</td>
</tr>
<tr>
<td></td>
<td><strong>self</strong></td>
<td>instance to which this method is bound</td>
</tr>
<tr>
<td>function</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this function was defined</td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong></td>
<td>qualified name</td>
</tr>
<tr>
<td></td>
<td><strong>code</strong></td>
<td>code object containing compiled function</td>
</tr>
<tr>
<td></td>
<td><strong>defaults</strong></td>
<td>tuple of any default values for positional parameters</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>kwdefaults</strong></td>
<td>mapping of any default values for keyword-only parameters</td>
<td></td>
</tr>
<tr>
<td><strong>globals</strong></td>
<td>global namespace in which this function was defined</td>
<td></td>
</tr>
<tr>
<td><strong>annotations</strong></td>
<td>mapping of parameters names to annotations; &quot;return&quot; key is reserved for return</td>
<td></td>
</tr>
<tr>
<td>traceback</td>
<td>tb_frame frame object at this level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tb_lasti index of last attempted instruction in bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tb_lineno current line number in Python source code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tb_next next inner traceback object (called by this level)</td>
<td></td>
</tr>
<tr>
<td>frame</td>
<td>f_back next outer frame object (this frame's caller)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_builtins builtins namespace seen by this frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_code code object being executed in this frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_globals global namespace seen by this frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_lasti index of last attempted instruction in bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_lineno current line number in Python source code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_locals local namespace seen by this frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_trace tracing function for this frame, or None</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>co_argcount number of arguments (not including keyword-only arguments, *, ** args)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_code string of raw compiled bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_cellvars tuple of names of cell variables (referenced by containing scopes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_consts tuple of constants used in the bytecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_filename name of file in which this code object was created</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_firstlineno number of first line in Python source code</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>co_flags</td>
<td>Bitmap of CO_* flags, read more here</td>
<td></td>
</tr>
<tr>
<td>co_lnotab</td>
<td>Encoded mapping of line numbers to bytecode indices</td>
<td></td>
</tr>
<tr>
<td>co_freevars</td>
<td>Tuple of names of free variables (referenced via a function's closure)</td>
<td></td>
</tr>
<tr>
<td>co_kwonlyargcount</td>
<td>Number of keyword only arguments (not including **arg)</td>
<td></td>
</tr>
<tr>
<td>co_name</td>
<td>Name with which this code object was defined</td>
<td></td>
</tr>
<tr>
<td>co_names</td>
<td>Tuple of names of local variables</td>
<td></td>
</tr>
<tr>
<td>co_nlocals</td>
<td>Number of local variables</td>
<td></td>
</tr>
<tr>
<td>co_stacksize</td>
<td>Virtual machine stack space required for execution of this code object</td>
<td></td>
</tr>
<tr>
<td>co_varnames</td>
<td>Tuple of names of arguments and local variables</td>
<td></td>
</tr>
<tr>
<td>generator</td>
<td><strong>name</strong>: name</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong>: qualified name</td>
<td></td>
</tr>
<tr>
<td>gi_frame</td>
<td>Frame</td>
<td></td>
</tr>
<tr>
<td>gi_running</td>
<td>Is the generator running?</td>
<td></td>
</tr>
<tr>
<td>gi_code</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>gi_yieldfrom</td>
<td>Object being iterated by <code>yield from</code></td>
<td></td>
</tr>
<tr>
<td>coroutine</td>
<td><strong>name</strong>: name</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong>: qualified name</td>
<td></td>
</tr>
<tr>
<td>cr_asyncargs</td>
<td>Object being awaited on, or None</td>
<td></td>
</tr>
<tr>
<td>cr_frame</td>
<td>Frame</td>
<td></td>
</tr>
<tr>
<td>cr_running</td>
<td>Is the coroutine running?</td>
<td></td>
</tr>
<tr>
<td>cr_code</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>cr_origin</td>
<td>Where coroutine was created, or None</td>
<td></td>
</tr>
</tbody>
</table>

* sys.set_coroutine_origin_t
<table>
<thead>
<tr>
<th>builtin</th>
<th><strong>doc</strong></th>
<th>documentation string</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>original name of this function or method</td>
</tr>
<tr>
<td></td>
<td><strong>qualname</strong></td>
<td>qualified name</td>
</tr>
<tr>
<td></td>
<td><strong>self</strong></td>
<td>instance to which a method is bound</td>
</tr>
</tbody>
</table>

*Changed in version 3.5:* Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

*Changed in version 3.7:* Add `cr_origin` attribute to coroutines.

**inspect.getmembers**(object[, predicate])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional `predicate` argument is supplied, only members for which the predicate returns a true value are included.

**Note:** `getmembers()` will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass’ custom `__dir__()`.  

**inspect.getmodulename**(path)

Return the name of the module named by the file `path`, without including the names of enclosing packages. The file extension is checked against all of the entries in `importlib.machinery.all_suffixes()`. If it matches, the final path component is returned with the extension removed. Otherwise, None is returned.

Note that this function only returns a meaningful name for actual
Python modules - paths that potentially refer to Python packages will still return `None`.

*Changed in version 3.3:* The function is based directly on `importlib`.

```python
inspect.ismodule(object)
Return true if the object is a module.
```

```python
inspect.isclass(object)
Return true if the object is a class, whether built-in or created in Python code.
```

```python
inspect.ismethod(object)
Return true if the object is a bound method written in Python.
```

```python
inspect.isfunction(object)
Return true if the object is a Python function, which includes functions created by a `lambda` expression.
```

```python
inspect.isgeneratorfunction(object)
Return true if the object is a Python generator function.
```

```python
inspect.isgenerator(object)
Return true if the object is a generator.
```

```python
inspect.iscoroutinefunction(object)
Return true if the object is a coroutine function (a function defined with an `async def` syntax).
```

*New in version 3.5.*

```python
inspect.iscoroutine(object)
```
Return true if the object is a coroutine created by an `async def` function.

*New in version 3.5.*

`inspect.isawaitable(object)`
Return true if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```python
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

*New in version 3.5.*

`inspect.isasyncgenfunction(object)`
Return true if the object is an asynchronous generator function, for example:

```python
>>> async def agen():
    ...    yield 1
    ...
>>> inspect.isasyncgenfunction(agen)
True
```

*New in version 3.6.*

`inspect.isasyncgen(object)`
Return true if the object is an asynchronous generator iterator created by an asynchronous generator function.

New in version 3.6.

inspect.istraceback(object)
Return true if the object is a traceback.

inspect.isframe(object)
Return true if the object is a frame.

inspect.iscode(object)
Return true if the object is a code.

inspect.isbuiltin(object)
Return true if the object is a built-in function or a bound built-in method.

inspect.isroutine(object)
Return true if the object is a user-defined or built-in function or method.

inspect.isabstract(object)
Return true if the object is an abstract base class.

inspect.ismethoddescriptor(object)
Return true if the object is a method descriptor, but not if ismethod(), isclass(), isfunction() or isbuiltin() are true.

This, for example, is true of int.__add__. An object passing this test has a __get__() method but not a __set__() method, but beyond that the set of attributes varies. A __name__ attribute is
usually sensible, and __doc__ often is.

Methods implemented via descriptors that also pass one of the other tests return false from the ismethoddescriptor() test, simply because the other tests promise more – you can, e.g., count on having the __func__ attribute (etc) when an object passes ismethod().

inspect.isdatadescriptor(object)
Return true if the object is a data descriptor.

Data descriptors have both a __get__ and a __set__ method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have __name__ and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.isgetsetdescriptor(object)
Return true if the object is a getset descriptor.

**CPython implementation detail:** getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

inspect.ismemberdescriptor(object)
Return true if the object is a member descriptor.

**CPython implementation detail:** Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this
method will always return False.
Retrieving source code

inspect.getdoc(object)

Get the documentation string for an object, cleaned up with cleandoc(). If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

*Changed in version 3.5:* Documentation strings are now inherited if not overridden.

inspect.getcomments(object)

Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return None. This could happen if the object has been defined in C or the interactive shell.

inspect.getfile(object)

Return the name of the (text or binary) file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

inspect.getmodule(object)

Try to guess which module an object was defined in.

inspect.getsourcefile(object)

Return the name of the Python source file in which an object was defined. This will fail with a TypeError if the object is a built-in
module, class, or function.

inspect.\texttt{getsourcelines}(\textit{object})
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An \texttt{OSError} is raised if the source code cannot be retrieved.

\textit{Changed in version 3.3:} \texttt{OSError} is raised instead of \texttt{IOError}, now an alias of the former.

inspect.\texttt{getsource}(\textit{object})
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An \texttt{OSError} is raised if the source code cannot be retrieved.

\textit{Changed in version 3.3:} \texttt{OSError} is raised instead of \texttt{IOError}, now an alias of the former.

inspect.\texttt{cleandoc}(\textit{doc})
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.
Introspecting callables with the Signature object

*New in version 3.3.*

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the `signature()` function.

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of Python callables, from plain functions and classes to `functools.partial()` objects.

Raises `ValueError` if no signature can be provided, and `TypeError` if that type of object is not supported.
A slash(/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

New in version 3.5: follow_wrapped parameter. Pass False to get a signature of callable specifically (callable.__wrapped__ will not be used to unwrap decorated callables.)

Note: Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

class inspect.Signature(parameters=None, *, return_annotation=Signature.empty)

A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a Parameter object in its parameters collection.

The optional parameters argument is a sequence of Parameter objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional return_annotation argument, can be an arbitrary Python object, is the “return” annotation of the callable.

Signature objects are immutable. Use Signature.replace() to make a modified copy.

Changed in version 3.5: Signature objects are picklable and
hashable.

empty
A special class-level marker to specify absence of a return annotation.

parameters
An ordered mapping of parameters’ names to the corresponding Parameter objects. Parameters appear in strict definition order, including keyword-only parameters.

Changed in version 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation
The “return” annotation for the callable. If the callable has no “return” annotation, this attribute is set to Signature.empty.

bind(*args, **kwargs)
Create a mapping from positional and keyword arguments to parameters. Returns BoundArguments if *args and **kwargs match the signature, or raises a TypeError.

bind_partial(*args, **kwargs)
Works the same way as Signature.bind(), but allows the omission of some required arguments (mimics functools.partial() behavior.) Returns BoundArguments, or raises a TypeError if the passed arguments do not match the signature.

replace(*[, parameters][, return_annotation])
Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different parameters and/or return_annotation to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in Signature.empty.

```python
>>> def test(a, b):
...   pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new")
>>> str(new_sig)
'(a, b) -> 'new return anno''
```

classmethod `from_callable(obj, *, follow_wrapped=True)`
Return a Signature (or its subclass) object for a given callable obj. Pass follow_wrapped=False to get a signature of obj without unwrapping its __wrapped__ chain.

This method simplifies subclassing of Signature:

```python
class MySignature(Signature):
  pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

*New in version 3.5.*

class inspect.**Parameter**(name, kind, *,
default=Parameter.empty, annotation=Parameter.empty)
Parameter objects are immutable. Instead of modifying a Parameter object, you can use Parameter.replace() to create a modified copy.
*Changed in version 3.5:* Parameter objects are picklable and hashable.

**empty**
A special class-level marker to specify absence of default values and annotations.

**name**
The name of the parameter as a string. The name must be a valid Python identifier.

**CPython implementation detail:** CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions. *Changed in version 3.6:* These parameter names are exposed by this module as names like `implicit0`.

**default**
The default value for the parameter. If the parameter has no default value, this attribute is set to `Parameter.empty`.

**annotation**
The annotation for the parameter. If the parameter has no annotation, this attribute is set to `Parameter.empty`.

**kind**
Describes how argument values are bound to the parameter. Possible values (accessible via `Parameter`, like `Parameter.KEYWORD_ONLY`):

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value must be supplied as</td>
</tr>
<tr>
<td><strong>POSITIONAL_ONLY</strong></td>
<td>a positional argument. Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>POSITIONAL_OR_KEYWORD</strong></td>
<td>Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)</td>
</tr>
<tr>
<td><strong>VAR_POSITIONAL</strong></td>
<td>A tuple of positional arguments that aren’t bound to any other parameter. This corresponds to a <code>*args</code> parameter in a Python function definition.</td>
</tr>
<tr>
<td><strong>KEYWORD_ONLY</strong></td>
<td>Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a <code>*</code> or <code>*args</code> entry in a Python function definition.</td>
</tr>
<tr>
<td></td>
<td>A dict of keyword arguments that aren’t bound to any other parameter. This</td>
</tr>
</tbody>
</table>
VAR\_KEYWORD

 corresponds to a **kwargs parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```python
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...         param.default is param)
...         print('Parameter:', param)
Parameter: c
```

**replace(***[name][, kind][, default][, annotation]*)**

Create a new Parameter instance based on the instance replaced was invoked on. To override a Parameter attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass Parameter.empty.

```python
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY)
>>> str(param)
'foo=42'

>>> str(param.replace())  # Will create a shallow copy
'foo=42'

>>> str(param.replace(default=Parameter.empty))
'foo=42'
```
"foo: 'spam'"

Changed in version 3.4: In Python 3.3 Parameter objects were allowed to have name set to None if their kind was set to POSITIONAL_ONLY. This is no longer permitted.

class inspect.BoundArguments

Result of a Signature.bind() or Signature.bind_partial() call. Holds the mapping of arguments to the function’s parameters.

arguments

An ordered, mutable mapping (collections.OrderedDict) of parameters’ names to arguments’ values. Contains only explicitly bound arguments. Changes in arguments will reflect in args and kwargs.

Should be used in conjunction with Signature.parameters for any argument processing purposes.

Note: Arguments for which Signature.bind() or Signature.bind_partial() relied on a default value are skipped. However, if needed, use BoundArguments.apply_defaults() to add them.

args

A tuple of positional arguments values. Dynamically computed from the arguments attribute.

kwargs

A dict of keyword arguments values. Dynamically computed
from the arguments attribute.

**signature**
A reference to the parent Signature object.

**apply_defaults()**
Set default values for missing arguments.

For variable-positional arguments (*args) the default is an empty tuple.

For variable-keyword arguments (**kwargs) the default is an empty dict.

```python
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

*New in version 3.5.*

The **args** and **kwargs** properties can be used to invoke functions:

```python
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

See also:

**PEP 362** - Function Signature Object.
The detailed specification, implementation details and examples.
Classes and functions

`inspect.getclasstree(classes, unique=False)`

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the `unique` argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.getargspec(func)`

Get the names and default values of a Python function’s parameters. A `named tuple` `ArgSpec(args, varargs, keywords, defaults)` is returned. `args` is a list of the parameter names. `varargs` and `keywords` are the names of the `*` and `**` parameters or `None`. `defaults` is a tuple of default argument values or `None` if there are no default arguments; if this tuple has `n` elements, they correspond to the last `n` elements listed in `args`.

*Deprecated since version 3.0:* Use `getfullargspec()` for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use `signature()` and `Signature Object`, which provide a more structured introspection API for callables.

`inspect.getfullargspec(func)`

Get the names and default values of a Python function’s
parameters. A named tuple is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)
```

`args` is a list of the positional parameter names. `varargs` is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. `varkw` is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. `defaults` is an `n`-tuple of default argument values corresponding to the last `n` positional parameters, or `None` if there are no such defaults defined. `kwonlyargs` is a list of keyword-only parameter names in declaration order. `kwonlydefaults` is a dictionary mapping parameter names from `kwonlyargs` to the default values used if no argument is supplied. `annotations` is a dictionary mapping parameter names to annotations. The special key "return" is used to report the function return value annotation (if any).

Note that `signature()` and `Signature Object` provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

*Changed in version 3.4:* This function is now based on `signature()`, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

*Changed in version 3.6:* This method was previously documented as deprecated in favour of `signature()` in Python 3.5, but that decision has been reversed in order to restore a clearly supported
standard interface for single-source Python 2/3 code migrating away from the legacy `getargspec()` API.

*Changed in version 3.7:* Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

`inspect.getargvalues(frame)`

Get information about arguments passed into a particular frame. A named tuple `ArgInfo(args, varargs, keywords, locals)` is returned. `args` is a list of the argument names. `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`. `locals` is the locals dictionary of the given frame.

Note: This function was inadvertently marked as deprecated in Python 3.5.

`inspect.formatargspec(args[, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations[, formatarg, formatvarargs, formatvarkw, formatvalue, formatreturns, formatannotations]]))`

Format a pretty argument spec from the values returned by `getfullargspec()`.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwonlyargs`, `kwonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.
For example:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

*Deprecated since version 3.5:* Use `signature()` and `Signature Object`, which provide a better introspecting API for callables.

```
inspect.formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue])
```

Format a pretty argument spec from the four values returned by `getargvalues()`. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

**Note:** This function was inadvertently marked as deprecated in Python 3.5.

```
inspect.getmro(cls)
```

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

```
inspect.getcallargs(func, *args, **kwds)
```

Bind the `args` and `kwds` to the argument names of the Python function or method `func`, as if it was called with them. For bound
methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the * and ** arguments, if any) to their values from `args` and `kwds`. In case of invoking `func` incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```python
def f(a, b=1, *pos, **named):
    ...
    pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named':
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named':
True
>>> getcallargs(f)
Traceback (most recent call last):
... AttributeError: f() missing 1 required positional argument:
```

New in version 3.2.

Deprecation note since version 3.5: Use `Signature.bind()` and `Signature.bind_partial()` instead.

`inspect.getclosurevars(func)`

Get the mapping of external name references in a Python function or method `func` to their current values. A named tuple `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. `nonlocals` maps referenced names to lexical closure variables, `globals` to the function’s module globals and `builtins` to the builtins visible from the function body. `unbound` is
the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

**TypeError** is raised if *func* is not a Python function or method.

*New in version 3.3.*

```python
inspect.unwrap(func, *, stop=Non)
```

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

*stop* is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, `signature()` uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

**ValueError** is raised if a cycle is encountered.

*New in version 3.4.*
The interpreter stack

When the following functions return “frame records,” each record is a named tuple `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

*Changed in version 3.5:* Return a named tuple instead of a tuple.

**Note:** Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python’s optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```python
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
```
If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional `context` argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.getframeinfo(frame, context=1)`

Get information about a frame or traceback object. A named tuple `Traceback(filename, lineno, function, code_context, index)` is returned.

`inspect.getouterframes(frame, context=1)`

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of `frame`. The first entry in the returned list represents `frame`; the last entry represents the outermost call on `frame`'s stack.

* Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.getinnerframes(traceback, context=1)`

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of `frame`. The first entry in the list represents `traceback`; the last entry represents where the exception was raised.

* Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.
inspect.currentframe()
Return the frame object for the caller’s stack frame.

**CPython implementation detail:** This function relies on Python stack frame support in the interpreter, which isn’t guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns None.

inspect.stack(context=1)
Return a list of frame records for the caller’s stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

*Changed in version 3.5:* A list of named tuples FrameInfo(frame, filename, lineno, function, code_context, index) is returned.

inspect.trace(context=1)
Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

*Changed in version 3.5:* A list of named tuples FrameInfo(frame, filename, lineno, function, code_context, index) is returned.
Fetching attributes statically

Both `getattr()` and `hasattr()` can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()`, `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. `getattr_static()` has the same signature as `getattr()` but avoids executing code when it fetches attributes.

```
inspect.getattr_static(obj, attr, default=None)
```

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`. Note: this function may not be able to retrieve all attributes that `getattr` can fetch (like dynamically created attributes) and may find attributes that `getattr` can’t (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance `__dict__` is shadowed by another member (for example a property) then this function will be unable to find instance members.

*New in version 3.2.*

`getattr_static()` does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.
You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```python
# example code for resolving the builtin descriptor
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```
Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. `getgeneratorstate()` allows the current state of a generator to be determined easily.

```
inspect.getgeneratorstate(generator)
```

Get current state of a generator-iterator.

Possible states are:

- **GEN_CREATED**: Waiting to start execution.
- **GEN_RUNNING**: Currently being executed by the interpreter.
- **GEN_SUSPENDED**: Currently suspended at a yield expression.
- **GEN_CLOSED**: Execution has completed.

*New in version 3.2.*

```
inspect.getcoroutinestate(coroutine)
```

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- **CORO_CREATED**: Waiting to start execution.
- **CORO_RUNNING**: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

New in version 3.5.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

```python
inspect.getgeneratorlocals(generator)
```

Get the mapping of live local variables in `generator` to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling `locals()` in the body of the generator, and all the same caveats apply.

If `generator` is a `generator` with no currently associated frame, then an empty dictionary is returned. `TypeError` is raised if `generator` is not a Python generator object.

**CPython implementation detail:** This function relies on the generator exposing a Python stack frame for introspection, which isn’t guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

New in version 3.3.

```python
inspect.getcoroutinelocals(coroutine)
```

This function is analogous to `getgeneratorlocals()`, but works for coroutine objects created by `async def` functions.

New in version 3.5.
Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

- `inspect.CO_OPTIMIZED`
  The code object is optimized, using fast locals.

- `inspect.CO_NEWLOCALS`
  If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

- `inspect.CO_VARARGS`
  The code object has a variable positional parameter (*args-like).

- `inspect.CO_VARKEYWORDS`
  The code object has a variable keyword parameter (**kwargs-like).

- `inspect.CO_NESTED`
  The flag is set when the code object is a nested function.

- `inspect.CO_GENERATOR`
  The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

- `inspect.CO_NOFREE`
  The flag is set if there are no free or cell variables.

- `inspect.CO_COROUTINE`
  The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See PEP
492 for more details.

*New in version 3.5.*

**inspect.CO_ITERABLE_COROUTINE**

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield` from coroutine objects. See PEP 492 for more details.

*New in version 3.5.*

**inspect.CO_ASYNC_GENERATOR**

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See PEP 525 for more details.

*New in version 3.6.*

**Note:** The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It’s recommended to use public APIs from the `inspect` module for any introspection needs.
Command Line Interface

The `inspect` module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

`--details`

Print information about the specified object rather than the source code
site — Site-specific configuration hook

Source code: Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter’s -S option.

Importing this module will append site-specific paths to the module search path and add a few builtins, unless -S was used. In that case, this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the site.main() function.

*Changed in version 3.3:* Importing the module used to trigger paths manipulation even when using -S.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/pythonX.Y/site-packages (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

*Changed in version 3.5:* Support for the “site-python” directory has been removed.

If a file named “pyvenv.cfg” exists one directory above sys.executable,
sys.prefix and sys.exec_prefix are set to that directory and it is also checked for site-packages (sys.base_prefix and sys.base_exec_prefix will always be the “real” prefixes of the Python installation). If “pyvenv.cfg” (a bootstrap configuration file) contains the key “ includesystem-site-packages” set to anything other than “false” (case-insensitive), the system-level prefixes will still also be searched for site-packages; otherwise they won’t.

A path configuration file is a file whose name has the form name.pth and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to sys.path. Non-existing items are never added to sys.path, and no check is made that the item refers to a directory rather than a file. No item is added to sys.path more than once. Blank lines and lines beginning with # are skipped. Lines starting with import (followed by space or tab) are executed.

For example, suppose sys.prefix and sys.exec_prefix are set to /usr/local. The Python X.Y library is then installed in /usr/local/lib/pythonX.Y. Suppose this has a subdirectory /usr/local/lib/pythonX.Y/site-packages with three subsubdirectories, foo, bar and spam, and two path configuration files, foo.pth and bar.pth. Assume foo.pth contains the following:

```python
# foo package configuration
foo
bar
bletch
```

and bar.pth contains:
# bar package configuration

bar

Then the following version-specific directories are added to `sys.path`, in this order:

```bash
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn’t exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an `ImportError` or its subclass exception, and the exception’s name attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from `sitecustomize` is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if `ENABLE_USER_SITE` is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. If this import fails with an `ImportError` or its subclass exception, and the exception’s name
attribute equals to 'usercustomize', it is silently ignored.

Note that for some non-Unix systems, sys.prefix and sys.exec_prefix are empty, and the path manipulations are skipped; however the import of sitecustomize and usercustomize is still attempted.
Readline configuration

On systems that support readline, this module will also import and configure the rlcompleter module, if Python is started in interactive mode and without the -S option. The default behavior is enable tab-completion and to use ~/.python_history as the history save file. To disable it, delete (or override) the sys.__interactivehook__ attribute in your sitecustomize or usercustomize module or your PYTHONSTARTUP file.

*Changed in version 3.4:* Activation of rlcompleter and history was made automatic.
Module contents

site.PREFIXES

A list of prefixes for site-packages directories.

site.ENABLE_USER_SITE

Flag showing the status of the user site-packages directory. True means that it is enabled and was added to `sys.path`. False means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). None means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

site.USER_SITE

Path to the user site-packages for the running Python. Can be None if `getusersitepackages()` hasn’t been called yet. Default value is `~/.local/lib/pythonX.Y/site-packages` for UNIX and non-framework Mac OS X builds, `~/Library/Python/X.Y/lib/python/site-packages` for Mac framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

site.USER_BASE

Path to the base directory for the user site-packages. Can be None if `getuserbase()` hasn’t been called yet. Default value is `~/.local` for UNIX and Mac OS X non-framework builds, `~/Library/Python/X.Y` for Mac framework builds, and `%APPDATA%\Python` for Windows. This value is used by Distutils
to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also PYTHONUSERBASE.

site.main()
Add all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the -S flag.

Changed in version 3.3: This function used to be called unconditionally.

site.addsitedir(sitedir, known_paths=None)
Add a directory to sys.path and process its .pth files. Typically used in sitecustomize or usercustomize (see above).

site.getsitepackages()
Return a list containing all global site-packages directories.

New in version 3.2.

site.getuserbase()
Return the path of the user base directory, USER_BASE. If it is not initialized yet, this function will also set it, respecting PYTHONUSERBASE.

New in version 3.2.

site.getusersitepackages()
Return the path of the user-specific site-packages directory, USER_SITE. If it is not initialized yet, this function will also set it, respecting PYTHONNOUSERSITE and USER_BASE.
New in version 3.2.

The `site` module also provides a way to get the user directories from the command line:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

If it is called without arguments, it will print the contents of `sys.path` on the standard output, followed by the value of `USER_BASE` and whether the directory exists, then the same thing for `USER_SITE`, and finally the value of `ENABLE_USER_SITE`.

```
--user-base
Print the path to the user base directory.

--user-site
Print the path to the user site-packages directory.
```

If both options are given, user base and user site will be printed (always in this order), separated by `os.pathsep`.

If any option is given, the script will exit with one of these values: 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

See also: PEP 370 – Per user site-packages directory
Custom Python Interpreters

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the code module. (The codeop module is lower-level, used to support compiling a possibly-incomplete chunk of Python code.)

The full list of modules described in this chapter is:

- **code** — Interpreter base classes
  - Interactive Interpreter Objects
  - Interactive Console Objects
- **codeop** — Compile Python code
The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

```python
class code.InteractiveInterpreter(locals=None):
    This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional locals argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '__name__' set to '__console__' and key '__doc__' set to None.
```

```python
class code.InteractiveConsole(locals=None, filename="<console>")
    Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar sys.ps1 and sys.ps2, and input buffering.
```

code.interact(banner=None, readfunc=None, local=None, exitmsg=None)
    Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets readfunc to be used as the InteractiveConsole.raw_input() method, if
provided. If \textit{local} is provided, it is passed to the \texttt{InteractiveConsole} constructor for use as the default namespace for the interpreter loop. The \texttt{interact()} method of the instance is then run with \textit{banner} and \textit{exitmsg} passed as the banner and exit message to use, if provided. The console object is discarded after use.

\textit{Changed in version 3.6: Added \textit{exitmsg} parameter.}

code.\texttt{compile\_command}(\texttt{source, filename=\"<input>\", symbol=\"single\")

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function \textit{almost} always makes the same decision as the real interpreter main loop.

\textit{source} is the source string; \textit{filename} is the optional filename from which source was read, defaulting to \texttt{\"<input>\"}; and \textit{symbol} is the optional grammar start symbol, which should be either \texttt{\"single\"} (the default) or \texttt{\"eval\"}.

Returns a code object (the same as \texttt{compile(source, filename, symbol)}) if the command is complete and valid; \texttt{None} if the command is incomplete; raises \texttt{SyntaxError} if the command is complete and contains a syntax error, or raises \texttt{OverflowError} or \texttt{ValueError} if the command contains an invalid literal.
Interactive Interpreter Objects

InteractiveInterpreter.runsource(source, filename="<input>", symbol="single")

Compile and run some source in the interpreter. Arguments are the same as for compile_command(); the default for filename is '<input>', and for symbol is 'single'. One several things can happen:

- The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method. runsource() returns False.
- The input is incomplete, and more input is required; compile_command() returned None. runsource() returns True.
- The input is complete; compile_command() returned a code object. The code is executed by calling the runcode() (which also handles run-time exceptions, except for SystemExit). runsource() returns False.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

InteractiveInterpreter.runcode(code)

 Execute a code object. When an exception occurs, showtraceback() is called to display a traceback. All exceptions are caught except SystemExit, which is allowed to propagate.

A note about KeyboardInterrupt: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.
InteractiveInterpreter. `showSyntaxError(filename=None)`
Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If `filename` is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `<string>` when reading from a string. The output is written by the `write()` method.

InteractiveInterpreter. `showTraceback()`
Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

*Changed in version 3.5:* The full chained traceback is displayed instead of just the primary traceback.

InteractiveInterpreter. `write(data)`
Write a string to the standard error stream (sys.stderr). Derived classes should override this to provide the appropriate output handling as needed.
Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.interact(banner=None, exitmsg=None)`

Closely emulate the interactive Python console. The optional `banner` argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it’s so close!).

The optional `exitmsg` argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If `exitmsg` is not given or `None`, a default message is printed.

*Changed in version 3.4:* To suppress printing any banner, pass an empty string.

*Changed in version 3.6:* Print an exit message when exiting.

`InteractiveConsole.push(line)`

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter’s `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some
way (this is the same as runsource()).

InteractiveConsole.resetbuffer()
Remove any unhandled source text from the input buffer.

InteractiveConsole.raw_input(prompt="")
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, EOFError is raised. The base implementation reads from sys.stdin; a subclass may replace this with a different implementation.
**codeop — Compile Python code**

**Source code:** Lib/codeop.py

The `codeop` module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the `code` module. As a result, you probably don’t want to use the module directly; if you want to include such a loop in your program you probably want to use the `code` module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print ‘```’ or ‘...’ next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The `codeop` module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

```python
codeop.compile_command(source, filename=<input>, symbol="single")
```

Tries to compile `source`, which should be a string of Python code and return a code object if `source` is valid Python code. In that case, the filename attribute of the code object will be `filename`, which defaults to `<input>`. Returns `None` if `source` is *not* valid Python code, but is a prefix of valid Python code.

If there is a problem with `source`, an exception will be raised.
SyntaxError is raised if there is invalid Python syntax, and OverflowError or ValueError if there is an invalid literal.

The symbol argument determines whether source is compiled as a statement ("single", the default) or as an expression ("eval"). Any other value will cause ValueError to be raised.

Note: It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

class codeop.Compile
Instances of this class have __call__() methods identical in signature to the built-in function compile(), but with the difference that if the instance compiles program text containing a __future__ statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.

class codeop.CommandCompiler
Instances of this class have __call__() methods identical in signature to compile_command(); the difference is that if the instance compiles program text containing a __future__ statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.
Importing Modules

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

- **zipimport** — Import modules from Zip archives
  - zipimporter Objects
  - Examples
- **pkgutil** — Package extension utility
- **modulefinder** — Find modules used by a script
  - Example usage of ModuleFinder
- **runpy** — Locating and executing Python modules
- **importlib** — The implementation of import
  - Introduction
  - Functions
  - importlib.abc — Abstract base classes related to import
  - importlib.resources — Resources
  - importlib.machinery — Importers and path hooks
  - importlib.util — Utility code for importers
  - Examples
    - Importing programmatically
    - Checking if a module can be imported
    - Importing a source file directly
    - Setting up an importer
    - Approximating importlib.import_module()
**zipimport** — Import modules from Zip archives

This module adds the ability to import Python modules (*.py, *.pyc) and packages from ZIP-format archives. It is usually not needed to use the `zipimport` module explicitly; it is automatically used by the built-in `import` mechanism for `sys.path` items that are paths to ZIP archives.

Typically, `sys.path` is a list of directory names as strings. This module also allows an item of `sys.path` to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but only files `.py` and `.pyc` are available for import. ZIP import of dynamic modules (.pyd, .so) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn’t contain `.pyc` files, importing may be rather slow.

ZIP archives with an archive comment are currently not supported.

**See also:**

**PKZIP Application Note**

Documentation on the ZIP file format by Phil Katz, the creator of
the format and algorithms used.

**PEP 273 - Import Modules from Zip Archives**
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

**PEP 302 - New Import Hooks**
The PEP to add the import hooks that help this module work.

This module defines an exception:

```python
exception zipimport.ZipImportError
    Exception raised by zipimporter objects. It's a subclass of ImportError, so it can be caught as ImportError, too.
```
zipimporter Objects

zipimporter is the class for importing ZIP files.

class zipimport.zipimporter(archivepath)
    Create a new zipimporter instance. archivepath must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an archivepath of foo/bar.zip/lib will look for modules in the lib directory inside the ZIP file foo/bar.zip (provided that it exists).

    ZipImportError is raised if archivepath doesn't point to a valid ZIP archive.

find_module(fullname[, path])
    Search for a module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or None if it wasn't. The optional path argument is ignored—it's there for compatibility with the importer protocol.

get_code(fullname)
    Return the code object for the specified module. Raise ZipImportError if the module couldn't be found.

get_data(pathname)
    Return the data associated with pathname. Raise OSError if the file wasn't found.

    Changed in version 3.3: IOError used to be raised instead of OSError.
get_filename(fullname)
Return the value __file__ would be set to if the specified module was imported. Raise ZipImportError if the module couldn't be found.

New in version 3.1.

get_source(fullname)
Return the source code for the specified module. Raise ZipImportError if the module couldn't be found, return None if the archive does contain the module, but has no source for it.

is_package(fullname)
Return True if the module specified by fullname is a package. Raise ZipImportError if the module couldn't be found.

load_module(fullname)
Load the module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the imported module, or raises ZipImportError if it wasn't found.

archive
The file name of the importer's associated ZIP file, without a possible subpath.

prefix
The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The archive and prefix attributes, when combined with a slash, equal the original archivepath argument given to the
zipimporter constructor.
Examples

Here is an example that imports a module from a ZIP archive - note that the `zipimport` module is not explicitly used.

```
$ unzip -l example.zip
Archive:  example.zip
  Length  Date  Time  Name
  --------  ----  ----  ----
     8467  11-26-02 22:30  jwzthreading.py
  --------  ----  ----  ----
     8467
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file
>>> import jwzthreading
>>> jwzthreading.__file__
'exmaple.zip/jwzthreading.py'
```
pkgutil — Package extension utility

Source code: Lib/pkgutil.py

This module provides utilities for the import system, in particular package support.

class pkgutil.ModuleInfo(module_finder, name, ispkg)

A namedtuple that holds a brief summary of a module’s info.

New in version 3.6.

pkgutil.extend_path(path, name)

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package’s __init__.py:

```python
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package’s __path__ all subdirectories of directories on sys.path named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the name argument. This feature is similar to *.pth files (see the site module for more information), except that it doesn’t special-case lines starting with import. A *.pkg file is trusted at face value:
apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that sys.path is a sequence. Items of sys.path that are not strings referring to existing directories are ignored. Unicode items on sys.path that cause errors when used as filenames may cause this function to raise an exception (in line with os.path.isdir() behavior).

class pkgutil.ImpImporter(dirname=None)

PEP 302 Finder that wraps Python’s “classic” import algorithm.

If dirname is a string, a PEP 302 finder is created that searches that directory. If dirname is None, a PEP 302 finder is created that searches the current sys.path, plus any modules that are frozen or built-in.

Note that ImpImporter does not currently support being used by placement on sys.meta_path.

Deprecated since version 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in importlib.

class pkgutil.ImpLoader(fullname, file, filename, etc)

Loader that wraps Python’s “classic” import algorithm.

Deprecated since version 3.3: This emulation is no longer needed,
as the standard import mechanism is now fully PEP 302 compliant and available in `importlib`.

`pkgutil.find_loader(fullname)`
Retrieve a module loader for the given `fullname`.

This is a backwards compatibility wrapper around `importlib.util.find_spec()` that converts most failures to `ImportError` and only returns the loader rather than the full `ModuleSpec`.

*Changed in version 3.3:* Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

*Changed in version 3.4:* Updated to be based on PEP 451

`pkgutil.get_importer(path_item)`
Retrieve a finder for the given `path_item`.

The returned finder is cached in `sys.path_importer_cache` if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of `sys.path_hooks` is necessary.

*Changed in version 3.3:* Updated to be based directly on `importlib` rather than relying on the package internal PEP 302 import emulation.

`pkgutil.get_loader(module_or_name)`
Get a loader object for `module_or_name`.

If the module or package is accessible via the normal import
mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

*Changed in version 3.3:* Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

*Changed in version 3.4:* Updated to be based on PEP 451

pkgutil.**iter_importers**(*fullname=''*)

Yield finder objects for the given module name.

If fullname contains a '.', the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

*Changed in version 3.3:* Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.**iter_modules**(*path=None, prefix='''*)

Yields ModuleInfo for all submodules on path, or, if path is None, all top-level modules on sys.path.

path should be either None or a list of paths to look for modules in.
prefix is a string to output on the front of every module name on output.

**Note:** Only works for a finder which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for importlib.machinery.FileFinder and zipimport.zipimporter.

*Changed in version 3.3:* Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.walk_packages(path=None, prefix='', onerror=None)

Yields ModuleInfo for all modules recursively on path, or, if path is None, all accessible modules.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all packages (not all modules!) on the given path, in order to access the __path__ attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no onerror function is supplied, ImportError s are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ +

Note: Only works for a finder which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for importlib.machinery.FileFinder and zipimport.zipimporter.

Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.get_data(package, resource)

Get a resource from a package.

This is a wrapper for the loader get_data API. The package argument should be the name of a package, in standard module format (foo.bar). The resource argument should be in the form of a relative filename, using / as the path separator. The parent directory name .. is not allowed, and nor is a rooted name (starting with a /).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:
If the package cannot be located or loaded, or it uses a loader which does not support `get_data`, then `None` is returned. In particular, the loader for namespace packages does not support `get_data`. 
modulefinder — Find modules used by a script

Source code: Lib/modulefinder.py

This module provides a ModuleFinder class that can be used to determine the set of modules imported by a script. modulefinder.py can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

modulefinder. AddPackagePath(pkg_name, path)
Record that the package named pkg_name can be found in the specified path.

modulefinder. ReplacePackage(oldname, newname)
Allows specifying that the module named oldname is in fact the package named newname.

class modulefinder. ModuleFinder(path=None, debug=0, excludes=[], replace_paths=[])
This class provides run_script() and report() methods to determine the set of modules imported by a script. path can be a list of directories to search for modules; if not specified, sys.path is used. debug sets the debugging level; higher values make the class print debugging messages about what it's doing. excludes is a list of module names to exclude from the analysis. replace_paths is a list of (oldpath, newpath) tuples that will be replaced in module paths.
**report()**
Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

**run_script(pathname)**
Analyze the contents of the *pathname* file, which must contain Python code.

**modules**
A dictionary mapping module names to modules. See [Example usage of ModuleFinder](#).
Example usage of **ModuleFinder**

The script that is going to get analyzed later on (bacon.py):

```python
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```python
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%%s: ' % name, end='')
    print(',').join(list(mod.globalnames.keys())[:3])

print('-'*50)
print('Modules not imported:')
print('
'.join(finder.badmodules.keys()))
```

Sample output (may vary depending on the architecture):
Loaded modules:
_types:
copyreg: _inverted_registry,_slotnames,__all__
sre_compile: isstring,_sre,_optimize_unicode
_sre:
sre_constants: REPEAT_ONE,makedict,AT_END_LINE
sys:
re: __module__,finditer,_expand
itertools:
__main__: re,itertools,baconhameggs
sre_parse: _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types: __module__,IntType,TypeType
---------------------------------------------------

Modules not imported:
guido.python.ham
baconhameggs
The `runpy` module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is not a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a `runpy` function has returned. If that limitation is not acceptable for a given use case, `importlib` is likely to be a more suitable choice than this module.

The `runpy` module provides two functions:

```python
runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)
```

Execute the code of the specified module and return the resulting module globals dictionary. The module’s code is first located using the standard import mechanism (refer to PEP 302 for details) and then executed in a fresh module namespace.

The `mod_name` argument should be an absolute module name. If the module name refers to a package rather than a normal module,
then that package is imported and the __main__ submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument init_globals may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by run_module().

The special global variables __name__, __spec__, __file__, __cached__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

__name__ is set to run_name if this optional argument is not None, to mod_name + '.__main__' if the named module is a package and to the mod_name argument otherwise.

__spec__ will be set appropriately for the actually imported module (that is, __spec__.name will always be mod_name or mod_name + '.__main__', never run_name).

__file__, __cached__, __loader__ and __package__ are set as normal based on the module spec.

If the argument alter_sys is supplied and evaluates to True, then sys.argv[0] is updated with the value of __file__ and sys.modules[__name__] is updated with a temporary module object for the module being executed. Both sys.argv[0] and sys.modules[__name__] are restored to their original values
before the function returns.

Note that this manipulation of \texttt{sys} is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the \texttt{sys} module be left alone when invoking this function from threaded code.

\textbf{See also:} The \texttt{-m} option offering equivalent functionality from the command line.

\textit{Changed in version 3.1:} Added ability to execute packages by looking for a \texttt{__main__} submodule.

\textit{Changed in version 3.2:} Added \texttt{__cached__} global variable (see \texttt{PEP 3147}).

\textit{Changed in version 3.4:} Updated to take advantage of the module spec feature added by \texttt{PEP 451}. This allows \texttt{__cached__} to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as \texttt{__spec__.name}.

\texttt{runpy.run_path(file_path, init\_globals=None, run\_name=None)}

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a \texttt{__main__} module (e.g. a zipfile containing a top-level \texttt{__main__.py} file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of \texttt{sys.path}. The function then looks for and executes a \texttt{__main__} module.
using the updated path. Note that there is no special protection against invoking an existing __main__ entry located elsewhere on sys.path if there is no such module at the specified location.

The optional dictionary argument init_globals may be used to pre-populate the module’s globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by run_path().

The special global variables __name__, __spec__, __file__, __cached__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

__name__ is set to run_name if this optional argument is not None and to '<run_path>' otherwise.

If the supplied path directly references a script file (whether as source or as precompiled byte code), then __file__ will be set to the supplied path, and __spec__, __cached__, __loader__ and __package__ will all be set to None.

If the supplied path is a reference to a valid sys.path entry, then __spec__ will be set appropriately for the imported __main__ module (that is, __spec__.name will always be __main__). __file__, __cached__, __loader__ and __package__ will be set as normal based on the module spec.

A number of alterations are also made to the sys module. Firstly, sys.path may be altered as described above. sys.argv[0] is
updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in `sys` are reverted before the function returns.

Note that, unlike `run_module()`, the alterations made to `sys` are not optional in this function as these adjustments are essential to allowing the execution of `sys.path` entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

**See also:** Interface options for equivalent functionality on the command line (`python path/to/script`).

*New in version 3.2.*

*Changed in version 3.4:* Updated to take advantage of the module spec feature added by [PEP 451](https://www.python.org/dev/peps/pep-0451/). This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid `sys.path` entry rather than being executed directly.

**See also:**

- **PEP 338** – Executing modules as scripts
  PEP written and implemented by Nick Coghlan.

- **PEP 366** – Main module explicit relative imports
  PEP written and implemented by Nick Coghlan.

- **PEP 451** – A ModuleSpec Type for the Import System
  PEP written and implemented by Eric Snow

Command line and environment - CPython command line details
The `importlib.import_module()` function
importlib — The implementation of import

New in version 3.1.

Source code: Lib/importlib/__init__.py
Introduction

The purpose of the `importlib` package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the `__import__() function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an `importer`) to participate in the import process.

See also:

**The import statement**

The language reference for the `import` statement.

**Packages specification**

Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in `sys.modules`).

**The __import__() function**

The `import` statement is syntactic sugar for this function.

**PEP 235**

Import on Case-Insensitive Platforms

**PEP 263**

Defining Python Source Code Encodings

**PEP 302**

New Import Hooks

**PEP 328**
Imports: Multi-Line and Absolute/Relative

**PEP 366**  
Main module explicit relative imports

**PEP 420**  
Implicit namespace packages

**PEP 451**  
A ModuleSpec Type for the Import System

**PEP 488**  
Elimination of PYO files

**PEP 489**  
Multi-phase extension module initialization

**PEP 552**  
Deterministic pycs

**PEP 3120**  
Using UTF-8 as the Default Source Encoding

**PEP 3147**  
PYC Repository Directories
Functions

importlib.__import__(name, globals=None, locals=None, fromlist=(), level=0)

An implementation of the built-in __import__() function.

Note: Programmatic importing of modules should use import_module() instead of this function.

importlib.import_module(name, package=None)

Import a module. The name argument specifies what module to import in absolute or relative terms (e.g. either pkg.mod or ..mod). If the name is specified in relative terms, then the package argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. import_module('..mod', 'pkg.subpkg') will import pkg.mod).

The import_module() function acts as a simplifying wrapper around importlib.__import__(). This means all semantics of the function are derived from importlib.__import__(). The most important difference between these two functions is that import_module() returns the specified package or module (e.g. pkg.mod), while __import__() returns the top-level package or module (e.g. pkg).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call invalidate_caches() in order for the new module to be noticed by the import system.
Changed in version 3.3: Parent packages are automatically imported.

```
importlib.find_loader(name, path=None)
```

Find the loader for a module, optionally within the specified path. If the module is in sys.modules, then `sys.modules[name].__loader__` is returned (unless the loader would be None or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. None is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to `path`.

New in version 3.3.

Changed in version 3.4: If `__loader__` is not set, raise `ValueError`, just like when the attribute is set to None.

Deprecated since version 3.4: Use `importlib.util.find_spec()` instead.

```
invalidate_caches()
```

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

New in version 3.3.
importlib.\texttt{reload}(\textit{module})

Reload a previously imported \textit{module}. The argument must be a
module object, so it must have been successfully imported before.
This is useful if you have edited the module source file using an
external editor and want to try out the new version without leaving
the Python interpreter. The return value is the module object (which
can be different if re-importing causes a different object to be
placed in \texttt{sys.modules}).

When \texttt{reload()} is executed:

- Python module's code is recompiled and the module-level
code re-executed, defining a new set of objects which are
bound to names in the module's dictionary by reusing the
\texttt{loader} which originally loaded the module. The \texttt{init}
function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only
reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to
any new or changed objects.
- Other references to the old objects (such as names external to
the module) are not rebound to refer to the new objects and
must be updated in each namespace where they occur if that
is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's
global variables) is retained. Redefinitions of names will override
the old definitions, so this is generally not a problem. If the new
version of a module does not define a name that was defined by
the old version, the old definition remains. This feature can be used
to the module's advantage if it maintains a global table or cache of
objects — with a \texttt{try} statement it can test for the table's presence
and skip its initialization if desired:

```python
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading `sys`, `__main__`, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (`module.name`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

*New in version 3.4.*

*Changed in version 3.7:* `ModuleNotFoundError` is raised when the module being reloaded lacks a `ModuleSpec`. 
importlib.abc – Abstract base classes related to import

Source code: Lib/importlib/abc.py

The importlib.abc module contains all of the core abstract base classes used by import. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
  +-- Finder (deprecated)
  |    +-- MetaPathFinder
  |    +-- PathEntryFinder
  +-- Loader
      +-- ResourceLoader
      +-- InspectLoader
          +-- ExecutionLoader
              +-- FileLoader
              +-- SourceLoader
```

class importlib.abc.Finder

An abstract base class representing a finder.

*Deprecated since version 3.3: Use MetaPathFinder or PathEntryFinder instead.*

abstractmethod **find_module**(fullname, path=None)

An abstract method for finding a loader for the specified module. Originally specified in PEP 302, this method was meant for use in sys.meta_path and in the path-based import
subsystem.

*Changed in version 3.4:* Returns `None` when called instead of raising `NotImplementedError`.

class importlib.abc.MetaPathFinder
An abstract base class representing a meta path finder. For compatibility, this is a subclass of `Finder`.

*New in version 3.3.*

**find_spec**(fullname, path, target=None)
An abstract method for finding a spec for the specified module. If this is a top-level import, `path` will be `None`. Otherwise, this is a search for a subpackage or module and `path` will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return.

*New in version 3.4.*

**find_module**(fullname, path)
A legacy method for finding a loader for the specified module. If this is a top-level import, `path` will be `None`. Otherwise, this is a search for a subpackage or module and `path` will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

If `find_spec()` is defined, backwards-compatible functionality is provided.

*Changed in version 3.4:* Returns `None` when called instead of
raising `NotImplementedError`. Can use `find_spec()` to provide functionality.

*Deprecated since version 3.4:* Use `find_spec()` instead.

**insecure_cache()**
An optional method which, when called, should invalidate any internal cache used by the finder. Used by `importlib.invalidate_caches()` when invalidating the caches of all finders on `sys.meta_path`.

*Changed in version 3.4:* Returns `None` when called instead of `NotImplemented`.

**class importlib.abc.PathEntryFinder**
An abstract base class representing a path entry finder. Though it bears some similarities to `MetaPathFinder`, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of `Finder` for compatibility reasons only.

*New in version 3.3.*

**find_spec(fullname, target=None)**
An abstract method for finding a spec for the specified module. The finder will search for the module only within the path entry to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return.

*New in version 3.4.*
**find_loader**(fullname)
A legacy method for finding a loader for the specified module. Returns a 2-tuple of (loader, portion) where portion is a sequence of file system locations contributing to part of a namespace package. The loader may be None while specifying portion to signify the contribution of the file system locations to a namespace package. An empty list can be used for portion to signify the loader is not part of a namespace package. If loader is None and portion is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If find_spec() is defined then backwards-compatible functionality is provided.

*Changed in version 3.4:* Returns (None, []) instead of raising NotImplementedError. Uses find_spec() when available to provide functionality.

*Deprecated since version 3.4:* Use find_spec() instead.

**find_module**(fullname)
A concrete implementation of Finder.find_module() which is equivalent to self.find_loader(fullname)[0].

*Deprecated since version 3.4:* Use find_spec() instead.

**invalidate_caches()**
An optional method which, when called, should invalidate any internal cache used by the finder. Used by PathFinder.invalidate_caches() when invalidating the caches of all cached finders.
class importlib.abc.Loader
An abstract base class for a loader. See PEP 302 for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by `importlib.abc.ResourceReader`.

*Changed in version 3.7:* Introduced the optional `get_resource_reader()` method.

create_module(spec)
A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

*New in version 3.4.*

*Changed in version 3.5:* Starting in Python 3.6, this method will not be optional when `exec_module()` is defined.

exec_module(module)
An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, `create_module()` must be defined.

*New in version 3.4.*

*Changed in version 3.6:* `create_module()` must also be defined.
**load_module**(fullname)

A legacy method for loading a module. If the module cannot be loaded, **ImportError** is raised, otherwise the loaded module is returned.

If the requested module already exists in **sys.modules**, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into **sys.modules** before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from **sys.modules**; modules already in **sys.modules** before the loader began execution should be left alone (see **importlib.util.module_for_loader()**).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded):

- **__name__**
  
  The name of the module.

- **__file__**
  
  The path to where the module data is stored (not set for built-in modules).

- **__cached__**
  
  The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).

- **__path__**
  
  A list of strings specifying the search path within a package. This attribute is not set on modules.
• **__package__**
  The parent package for the module/package. If the module is top-level then it has a value of the empty string. The `importlib.util.module_for_loader()` decorator can handle the details for **__package__**.

• **__loader__**
  The loader used to load the module. The `importlib.util.module_for_loader()` decorator can handle the details for **__package__**.

When `exec_module()` is available then backwards-compatible functionality is provided.

*Changed in version 3.4:* Raise `ImportError` when called instead of `NotImplementedError`. Functionality provided when `exec_module()` is available.

*Deprecated since version 3.4:* The recommended API for loading a module is `exec_module()` (and `create_module()`). Loaders should implement it instead of `load_module()`. The import machinery takes care of all the other responsibilities of `load_module()` when `exec_module()` is implemented.

`module_repr(module)`
A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default `repr()` will use the result of this method as appropriate.

*New in version 3.3.*

*Changed in version 3.4:* Made optional instead of an
abstractmethod.

*Deprecated since version 3.4:* The import machinery now takes care of this automatically.

class importlib.abc.ResourceReader
An abstract base class to provide the ability to read resources.

From the perspective of this ABC, a resource is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored in a e.g. zip file versus on the file system.

For any of methods of this class, a resource argument is expected to be a path-like object which represents conceptually just a file name. This means that no subdirectory paths should be included in the resource argument. This is because the location of the package the reader is for, acts as the “directory”. Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_loader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

*New in version 3.7.*
abstract method `open_resource(resource)`

Returns an opened, file-like object for binary reading of the resource.

If the resource cannot be found, `FileNotFoundException` is raised.

abstract method `resource_path(resource)`

Returns the file system path to the resource.

If the resource does not concretely exist on the file system, raise `FileNotFoundException`.

abstract method `is_resource(name)`

Returns True if the named name is considered a resource. `FileNotFoundException` is raised if name does not exist.

abstract method `contents()`

Returns an iterable of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.
class importlib.abc.ResourceLoader

An abstract base class for a loader which implements the optional PEP 302 protocol for loading arbitrary resources from the storage back-end.

*Deprecated since version 3.7:* This ABC is deprecated in favour of supporting resource loading through `importlib.abc.ResourceReader`.

*abstractmethod* get_data(path)

An abstract method to return the bytes for the data located at `path`. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. `OSError` is to be raised if the `path` cannot be found. The `path` is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

*Changed in version 3.4:* Raises `OSError` instead of `NotImplementedError`.

class importlib.abc.InspectLoader

An abstract base class for a loader which implements the optional PEP 302 protocol for loaders that inspect modules.

get_code(fullname)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an `ImportError` if loader cannot find the requested module.

*Note:* While the method has a default implementation, it is
suggested that it be overridden if possible for performance.

*Changed in version 3.4:* No longer abstract and a concrete implementation is provided.

**abstractmethod `get_source(fullname)`**

An abstract method to return the source of a module. It is returned as a text string using universal newlines, translating all recognized line separators into '\n' characters. Returns None if no source is available (e.g. a built-in module). Raises **ImportError** if the loader cannot find the module specified.

*Changed in version 3.4:* Raises **ImportError** instead of **NotImplementedError**.

**is_package(fullname)**

An abstract method to return a true value if the module is a package, a false value otherwise. **ImportError** is raised if the loader cannot find the module.

*Changed in version 3.4:* Raises **ImportError** instead of **NotImplementedError**.

**static `source_to_code(data, path='<string>')`**

Create a code object from Python source.

The `data` argument can be whatever the `compile()` function supports (i.e. string or bytes). The `path` argument should be the “path” to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`. 
New in version 3.4.

Changed in version 3.5: Made the method static.

**exec_module**(module)

Implementation of `Loader.exec_module()`.

New in version 3.4.

**load_module**(fullname)

Implementation of `Loader.load_module()`.

*Deprecated since version 3.4:* use `exec_module()` instead.

```python
class importlib.abc.ExecutionLoader
    An abstract base class which inherits from `InspectLoader` that, when implemented, helps a module to be executed as a script. The ABC represents an optional [PEP 302](https://www.python.org/dev/peps/pep-0302/) protocol.

    *abstractmethod* `get_filename`(fullname)
        An abstract method that is to return the value of `__file__` for the specified module. If no path is available, `ImportError` is raised.

        If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

        *Changed in version 3.4:* Raises `ImportError` instead of `NotImplementedError`.

    ```
class importlib.abc.FileLoader(fullname, path)
    An abstract base class which inherits from `ResourceLoader` and
ExecutionLoader, providing concrete implementations of ResourceLoader.get_data() and ExecutionLoader.get_filename().

The fullname argument is a fully resolved name of the module the loader is to handle. The path argument is the path to the file for the module.

New in version 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module(fullname)

Calls super's load_module().

Deprecated since version 3.4: Use Loader.exec_module() instead.

abstractmethod get_filename(fullname)

Returns path.

abstractmethod get_data(path)

Reads path as a binary file and returns the bytes from it.

class importlib.abc.SourceLoader

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both ResourceLoader and ExecutionLoader, requiring the implementation of:
- `ResourceLoader.get_data()`  
- `ExecutionLoader.get_filename()`  
  
  Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise `NotImplementedError`) causes the loader to only work with source code. Implementing the methods allows the loader to work with source and bytecode files; it does not allow for sourceless loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

**path_stats(path)**

Optional abstract method which returns a `dict` containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, `OSError` is raised.

*New in version 3.3.*

*Changed in version 3.4:* Raise `OSError` instead of `NotImplementedError`.

**path_mtime(path)**

Optional abstract method which returns the modification time for
the specified path.

*Deprecated since version 3.3:* This method is deprecated in favour of `path_stats()`. You don’t have to implement it, but it is still available for compatibility purposes. Raise `OSError` if the path cannot be handled.

*Changed in version 3.4:* Raise `OSError` instead of `NotImplementedError`.

**set_data(path, data)**

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (`errno.EACCES/PermissionError`), do not propagate the exception.

*Changed in version 3.4:* No longer raises `NotImplementedError` when called.

**get_code(fullname)**

Concrete implementation of `InspectLoader.get_code()`.

**exec_module(module)**

Concrete implementation of `Loader.exec_module()`.

*New in version 3.4.*

**load_module(fullname)**

Concrete implementation of `Loader.load_module()`.
 Deprecated since version 3.4: Use `exec_module()` instead.

**get_source**(fullname)
Concrete implementation of `InspectLoader.get_source()`.

**is_package**(fullname)
Concrete implementation of `InspectLoader.is_package()`. A module is determined to be a package if its file path (as provided by `ExecutionLoader.get_filename()`) is a file named `__init__` when the file extension is removed and the module name itself does not end in `__init__`. 
importlib.resources – Resources

Source code: Lib/importlib/resources.py

New in version 3.7.

This module leverages Python’s import system to provide access to resources within packages. If you can import a package, you can access resources within that package. Resources can be opened or read, in either binary or text mode.

Resources are roughly akin to files inside directories, though it’s important to keep in mind that this is just a metaphor. Resources and packages do not have to exist as physical files and directories on the file system.

Note: This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

Loaders that wish to support resource reading should implement a get_resource_reader(fullname) method as specified by importlib.abc.ResourceReader.

The following types are defined.
**importlib.resources.**Package

The Package type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a Package, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not None.

**importlib.resources.**Resource

This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

The following functions are available.

**importlib.resources.**open_binary**(package, resource)**

Open for binary reading the resource within package.

*package* is either a name or a module object which conforms to the Package requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns a `typing.BinaryIO` instance, a binary I/O stream open for reading.

**importlib.resources.**open_text**(package, resource, encoding='utf-8', errors='strict')**

Open for text reading the resource within package. By default, the resource is opened for reading as UTF-8.

*package* is either a name or a module object which conforms to the Package requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may
not have sub-resources (i.e. it cannot be a directory). encoding and errors have the same meaning as with built-in open().

This function returns a typing.TextIO instance, a text I/O stream open for reading.

importlib.resources.read_binary(package, resource)
Read and return the contents of the resource within package as bytes.

package is either a name or a module object which conforms to the Package requirements. resource is the name of the resource to open within package; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns the contents of the resource as bytes.

importlib.resources.read_text(package, resource, encoding='utf-8', errors='strict')
Read and return the contents of resource within package as a str.
By default, the contents are read as strict UTF-8.

package is either a name or a module object which conforms to the Package requirements. resource is the name of the resource to open within package; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). encoding and errors have the same meaning as with built-in open(). This function returns the contents of the resource as str.

importlib.resources.path(package, resource)
Return the path to the resource as an actual file system path. This function returns a context manager for use in a with statement. The context manager provides a pathlib.Path object.
Exiting the context manager cleans up any temporary file created when the resource needs to be extracted from e.g. a zip file.

*package* is either a name or a module object which conforms to the Package requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory).

**importlib.resources.is_resource**(*package*, *name*)

Return True if there is a resource named *name* in the package, otherwise False. Remember that directories are not resources! *package* is either a name or a module object which conforms to the Package requirements.

**importlib.resources.contents**(*package*)

Return an iterable over the named items within the package. The iterable returns str resources (e.g. files) and non-resources (e.g. directories). The iterable does not recurse into subdirectories.

*package* is either a name or a module object which conforms to the Package requirements.
importlib.machinery – Importers and path hooks

Source code: Lib/importlib/machinery.py

This module contains the various objects that help import find and load modules.

importlib.machinery.SOURCE_SUFFIXES
A list of strings representing the recognized file suffixes for source modules.

New in version 3.3.

importlib.machinery.DEBUG_BYTECODE_SUFFIXES
A list of strings representing the file suffixes for non-optimized bytecode modules.

New in version 3.3.

Deprecated since version 3.5: Use BYTECODE_SUFFIXES instead.

importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES
A list of strings representing the file suffixes for optimized bytecode modules.

New in version 3.3.

Deprecated since version 3.5: Use BYTECODE_SUFFIXES instead.

importlib.machinery.BYTECODE_SUFFIXES
A list of strings representing the recognized file suffixes for
bytecode modules (including the leading dot).

*New in version 3.3.*

*Changed in version 3.5:* The value is no longer dependent on `__debug__`.

`importlib.machinery.EXTENSION_SUFFIXES`
A list of strings representing the recognized file suffixes for extension modules.

*New in version 3.3.*

`importlib.machinery.all_suffixes()`
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, `inspect.getmodulename()`).

*New in version 3.3.*

`class importlib.machinery.BuiltinImporter`
An importer for built-in modules. All known built-in modules are listed in `sys.builtin_module_names`. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

*Changed in version 3.5:* As part of PEP 489, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`.
class importlib.machinery.FrozenImporter

An importer for frozen modules. This class implements the importlib.abc.MetaPathFinder and importlib.abc.InspectLoader ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

class importlib.machinery.WindowsRegistryFinder

Finder for modules declared in the Windows registry. This class implements the importlib.abc.MetaPathFinder ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

New in version 3.3.

Deprecated since version 3.6: Use site configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.PathFinder

A Finder for sys.path and package __path__ attributes. This class implements the importlib.abc.MetaPathFinder ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(fullname, path=None, target=None)

Class method that attempts to find a spec for the module specified by fullname on sys.path or, if defined, on path. For each path entry that is searched, sys.path_importer_cache is checked. If a non-false object is found then it is used as the path entry finder to look for the
module being searched for. If no entry is found in \texttt{sys.path_importer_cache}, then \texttt{sys.path_hooks} is searched for a finder for the path entry and, if found, is stored in \texttt{sys.path_importer_cache} along with being queried about the module. If no finder is ever found then \texttt{None} is both stored in the cache and returned.

\textit{New in version 3.4.}

\textit{Changed in version 3.5:} If the current working directory – represented by an empty string – is no longer valid then \texttt{None} is returned but no value is cached in \texttt{sys.path_importer_cache}.

\texttt{classmethod find\_module}(\texttt{fullname, path=\texttt{None}})

A legacy wrapper around \texttt{find\_spec()}.  

\textit{Deprecated since version 3.4:} Use \texttt{find\_spec()} instead.

\texttt{classmethod invalidate\_caches}()

Calls \texttt{importlib.abc.PathEntryFinder.invalidate\_caches()} on all finders stored in \texttt{sys.path_importer_cache} that define the method. Otherwise entries in \texttt{sys.path_importer_cache} set to \texttt{None} are deleted.

\textit{Changed in version 3.7:} Entries of \texttt{None} in \texttt{sys.path_importer_cache} are deleted.

\textit{Changed in version 3.4:} Calls objects in \texttt{sys.path\_hooks} with the current working directory for \texttt{''} (i.e. the empty string).

\texttt{class importlib.machinery.FileFinder(path,}
A concrete implementation of `importlib.abc.PathEntryFinder` which caches results from the file system.

The `path` argument is the directory for which the finder is in charge of searching.

The `loader_details` argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

*New in version 3.3.*

**path**

The path the finder will search in.

**find_spec(fullname, target=None)**

Attempt to find the spec to handle `fullname` within `path`.

*New in version 3.4.*
**find_loader**(*fullname*)

Attempt to find the loader to handle *fullname* within *path*.

**invalidate_caches()**

Clear out the internal cache.

**classmethod path_hook(**loader_details**)**

A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and `loader_details` indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

**class** `importlib.machinery.SourceFileLoader`(*fullname*, *path*)

A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

*New in version 3.3.*

**name**

The name of the module that this loader will handle.

**path**

The path to the source file.

**is_package**(*fullname*)

Return true if *path* appears to be for a package.

**path_stats**(*path*)
Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

**set_data**(path, data)
Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

**load_module**(name=None)
Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

*Deprecated since version 3.6:* Use `importlib.abc.Loader.exec_module()` instead.

**class**
importlib.machinery.**SourcelessFileLoader**(fullname, path)
A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

*New in version 3.3.*

**name**
The name of the module the loader will handle.

**path**
The path to the bytecode file.
is_package(fullname)

Determines if the module is a package based on path.

get_code(fullname)

Returns the code object for name created from path.

get_source(fullname)

Returns None as bytecode files have no source when this loader is used.

load_module(name=None)

Concrete implementation of importlib.abc.Loader.load_module() where specifying the name of the module to load is optional.

Deprecated since version 3.6: Use importlib.abc.Loader.exec_module() instead.

class importlib.machinery.ExtensionFileLoader(fullname, path)

A concrete implementation of importlib.abc.ExecutionLoader for extension modules.

The fullname argument specifies the name of the module the loader is to support. The path argument is the path to the extension module's file.

New in version 3.3.

name

Name of the module the loader supports.
path
  Path to the extension module.

create_module(spec)
  Creates the module object from the given specification in accordance with PEP 489.

  New in version 3.5.

exec_module(module)
  Initializes the given module object in accordance with PEP 489.

  New in version 3.5.

is_package(fullname)
  Returns True if the file path points to a package's __init__ module based on EXTENSION_SUFFIXES.

get_code(fullname)
  Returns None as extension modules lack a code object.

get_source(fullname)
  Returns None as extension modules do not have source code.

get_filename(fullname)
  Returns path.

  New in version 3.4.

class importlib.machinery.ModuleSpec(name, loader, *, origin=None, loader_state=None, is_package=None)
  A specification for a module's import-system-related state. This is typically exposed as the module's __spec__ attribute. In the
descriptions below, the names in parentheses give the corresponding attribute available directly on the module object. E.g. 

```python
module.__spec__.origin == module.__file__.
```

Note however that while the values are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

*New in version 3.4.*

**name**

`( __name__ )`

A string for the fully-qualified name of the module.

**loader**

`( __loader__ )`

The loader to use for loading. For namespace packages this should be set to `None`.

**origin**

`( __file__ )`

Name of the place from which the module is loaded, e.g. “builtin” for built-in modules and the filename for modules loaded from source. Normally “origin” should be set, but it may be `None` (the default) which indicates it is unspecified (e.g. for namespace packages).

**submodule_search_locations**
(__path__)  
List of strings for where to find submodules, if a package (None otherwise).

**loader_state**  
Container of extra module-specific data for use during loading (or None).

**cached**  
(__cached__)  
String for where the compiled module should be stored (or None).

**parent**  
(__package__)  
(Read-only) Fully-qualified name of the package to which the module belongs as a submodule (or None).

**has_location**  
Boolean indicating whether or not the module’s “origin” attribute refers to a loadable location.
**importlib.util** – Utility code for importers

**Source code:** Lib/importlib/util.py

This module contains the various objects that help in the construction of an importer.

**importlib.util.MAGIC_NUMBER**

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider `importlib.abc.SourceLoader`.

*New in version 3.4.*

**importlib.util.cache_from_source**(path, debug_override=None, *, optimization=None)

Return the PEP 3147/PEP 488 path to the byte-compiled file associated with the source path. For example, if path is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`) if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised).

The `optimization` parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an `optimization` of `' '` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. None causes the interpreter’s optimization level to be used.
Any other value's string representation being used, so 
/foo/bar/baz.py with an optimization of 2 will lead to the 
bytecode path of /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc. The string representation of optimization can only 
be alphanumeric, else ValueError is raised.

The debug_override parameter is deprecated and can be used to 
override the system's value for __debug__. A True value is the 
equivalent of setting optimization to the empty string. A False 
value is the same as setting optimization to 1. If both 
debug_override an optimization are not None then TypeError 
is raised.

New in version 3.4.

Changed in version 3.5: The optimization parameter was added 
and the debug_override parameter was deprecated.

Changed in version 3.6: Accepts a path-like object.

importlib.util.source_from_cache(path)

Given the path to a PEP 3147 file name, return the associated 
source code file path. For example, if path is 
/foo/bar/__pycache__/baz.cpython-32.pyc the returned 
path would be /foo/bar/baz.py. path need not exist, however if 
it does not conform to PEP 3147 or PEP 488 format, a 
ValueError is raised. If sys.implementation.cache_tag is 
not defined, NotImplementedError is raised.

New in version 3.4.

Changed in version 3.6: Accepts a path-like object.
importlib.util.**decode_source**(source_bytes)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by importlib.abc.InspectLoader.get_source()).

*New in version 3.4.*

importlib.util.**resolve_name**(name, package)

Resolve a relative module name to an absolute one.

If `name` has no leading dots, then `name` is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the `package` argument is needed.

`ValueError` is raised if `name` is a relative module name but `package` is a false value (e.g. `None` or the empty string). `ValueError` is also raised a relative name would escape its containing package (e.g. requesting `.bacon` from within the `spam` package).

*New in version 3.3.*

importlib.util.**find_spec**(name, package=None)

Find the `spec` for a module, optionally relative to the specified `package` name. If the module is in `sys.modules`, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no spec is found.

If `name` is for a submodule (contains a dot), the parent module is automatically imported.
name and package work the same as for import_module().

New in version 3.4.

Changed in version 3.7: Raises ModuleNotFoundError instead of AttributeError if package is in fact not a package (i.e. lacks a __path__ attribute).

importlib.util.module_from_spec(spec)
Create a new module based on spec and spec.loader.create_module.

If spec.loader.create_module does not return None, then any pre-existing attributes will not be reset. Also, no AttributeError will be raised if triggered while accessing spec or setting an attribute on the module.

This function is preferred over using types.ModuleType to create a new module as spec is used to set as many import-controlled attributes on the module as possible.

New in version 3.5.

@importlib.util.module_for_loader
A decorator for importlib.abc.Loader.load_module() to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. load_module(self, module)) for which the second argument will be the module object to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the name of the module to be
loaded as expected for a loader. If the module is not found in sys.modules then a new one is constructed. Regardless of where the module came from, __loader__ set to self and __package__ is set based on what importlib.abc.InspectLoader.is_package() returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to sys.modules, then the module will be removed to prevent a partially initialized module from being in left in sys.modules. If the module was already in sys.modules then it is left alone.

*Changed in version 3.3:* __loader__ and __package__ are automatically set (when possible).

*Changed in version 3.4:* Set __name__, __loader__ __package__ unconditionally to support reloading.

*Deprecated since version 3.4:* The import machinery now directly performs all the functionality provided by this function.

@importlib.util.set_loader

A decorator for importlib.abc.Loader.load_module() to set the __loader__ attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. self) is what __loader__ should be set to.

*Changed in version 3.4:* Set __loader__ if set to None, as if the attribute does not exist.


*Deprecated since version 3.4:* The import machinery takes care of this automatically.

@importlib.util.set_package

A decorator for `importlib.abc.Loader.load_module()` to set the `__package__` attribute on the returned module. If `__package__` is set and has a value other than `None` it will not be changed.

*Deprecated since version 3.4:* The import machinery takes care of this automatically.

`importlib.util.spec_from_loader(name, loader, *, origin=None, is_package=None)`

A factory function for creating a `ModuleSpec` instance based on a loader. The parameters have the same meaning as they do for `ModuleSpec`. The function uses available `loader` APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

*New in version 3.4.*

`importlib.util.spec_from_file_location(name, location, *, loader=None, submodule_search_locations=None)`

A factory function for creating a `ModuleSpec` instance based on the path to a file. Missing information will be filled in on the spec by making use of `loader` APIs and by the implication that the module will be file-based.

*New in version 3.4.*

*Changed in version 3.6:* Accepts a path-like object.
importlib.util.**source_hash**(source_bytes)

Return the hash of source_bytes as bytes. A hash-based .pyc file embeds the source_hash() of the corresponding source file's contents in its header.

*New in version 3.7.*

**class** importlib.util.**LazyLoader**(loader)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define **exec_module()** as control over what module type is used for the module is required. For those same reasons, the loader's **create_module()** method must return None or a type for which its __class__ attribute can be mutated along with not using slots. Finally, modules which substitute the object placed into sys.modules will not work as there is no way to properly replace the module references throughout the interpreter safely; ValueError is raised if such a substitution is detected.

**Note:** For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

*New in version 3.5.*

*Changed in version 3.6:* Began calling **create_module()**, removing the compatibility warning for

**classmethod **factory**(loader)

A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```python
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory
finder = importlib.machinery.FileFinder(path, (]
```
Examples

Importing programmatically

To programmatically import a module, use `importlib.import_module()`.

```python
import importlib
itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use `importlib.util.find_spec()`.

```python
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```
Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```python
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)

# Optional; only necessary if you want to be able to import
# by name later.
sys.modules[module_name] = module
```

Setting up an importer

For deep customizations of import, you typically want to implement an importer. This means managing both the finder and loader side of things. For finders there are two flavours to choose from depending on your needs: a meta path finder or a path entry finder. The former is what you would put on `sys.meta_path` while the latter is what you create using a path entry hook on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                   importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location.
sys.path_hooks.append(SpamPathEntryFinder.path_hook)

Approximating importlib.import_module()

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of importlib.import_module() (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import.""
    absolute_name = importlib.util.resolve_name(name
try:
    return sys.modules[absolute_name]
except KeyError:
    pass

path = None
if '.' in absolute_name:
    parent_name, _, child_name = absolute_name.rpartition('.
    parent_module = import_module(parent_name)
    path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
else:
    msg = f'No module named {absolute_name!r}'
    raise ModuleNotFoundError(msg, name=absolute_name)
module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
Python Language Services

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

- **parser** — Access Python parse trees
  - Creating ST Objects
  - Converting ST Objects
  - Queries on ST Objects
  - Exceptions and Error Handling
  - ST Objects
  - Example: Emulation of compile()
- **ast** — Abstract Syntax Trees
  - Node classes
  - Abstract Grammar
  - ast Helpers
- **symtable** — Access to the compiler’s symbol tables
  - Generating Symbol Tables
  - Examining Symbol Tables
- **symbol** — Constants used with Python parse trees
- **token** — Constants used with Python parse trees
- **keyword** — Testing for Python keywords
- **tokenize** — Tokenizer for Python source
  - Tokenizing Input
  - Command-Line Usage
  - Examples
- **tabnanny** — Detection of ambiguous indentation
- **pyclbr** — Python class browser support
  - Function Objects
- **Class Objects**
  - `py_compile` — Compile Python source files
  - `compileall` — Byte-compile Python libraries
    - Command-line use
    - Public functions
  - `dis` — Disassembler for Python bytecode
    - Bytecode analysis
    - Analysis functions
    - Python Bytecode Instructions
    - Opcode collections
  - `pickletools` — Tools for pickle developers
    - Command line usage
    - Command line options
    - Programmatic Interface
The **parser** module provides an interface to Python’s internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

**Note:** From Python 2.5 onward, it’s much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the **ast** module.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the **parser** module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to *The Python Language Reference*. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the `expr()` or `suite()` functions, described below. The ST objects created by `sequence2st()` faithfully simulate those structures. Be aware that
the values of the sequences which are considered “correct” will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by \texttt{st2list()} or \texttt{st2tuple()} has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file \texttt{Include/graminit.h} and the Python module \texttt{symbol}. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword \texttt{if} in an \texttt{if_stmt}, are included in the node tree without any special treatment. For example, the \texttt{if} keyword is represented by the tuple \((1, 'if')\), where 1 is the numeric value associated with all \texttt{NAME} tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as \((1, 'if', 12)\), where the 12 represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the \texttt{if} keyword above is representative.
The various types of terminal symbols are defined in the C header file Include/token.h and the Python module token.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple “wrapper” class may be created in Python to hide the use of ST objects.

The parser module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

**See also:**

**Module symbol**
Useful constants representing internal nodes of the parse tree.

**Module token**
Useful constants representing leaf nodes of the parse tree and functions for testing node values.
Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the 'eval' and 'exec' forms.

**parser.**expr**(source)**

The **expr**() function parses the parameter **source** as if it were an input to **compile**(source, 'file.py', 'eval'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

**parser.**suite**(source)**

The **suite**() function parses the parameter **source** as if it were an input to **compile**(source, 'file.py', 'exec'). If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

**parser.**sequence2st**(sequence)**

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a **ParserError** exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to **compilest()**. This may indicate problems not related to syntax (such as a **MemoryError** exception), but may also be due to constructs such as the result of
parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.tuple2st(sequence)`

This is the same function as `sequence2st()`. This entry point is maintained for backward compatibility.
Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

```
parsing.parser.st2list(st, line_info=False, col_info=False)
```

This function accepts an ST object from the caller in st and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, `st2tuple()` should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token ends. This information is omitted if the flag is false or omitted.

```
parsing.parser.st2tuple(st, line_info=False, col_info=False)
```

This function accepts an ST object from the caller in st and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to `st2list()`.

If `line_info` is true, line number information will be included for all
terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

```
parser.compilest(st, filename='<syntax-tree>
```

The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in `exec()` or `eval()` functions. This function provides the interface to the compiler, passing the internal parse tree from `st` to the parser, using the source file name specified by the `filename` parameter. The default value supplied for `filename` indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a `SyntaxError` caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `SyntaxError` raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.
Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2st()`.

**parser.** `isexpr(st)`

When `st` represents an 'eval' form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compilest()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

**parser.** `issuite(st)`

This function mirrors `isexpr()` in that it reports whether an ST object represents an 'exec' form, commonly known as a “suite.” It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.
Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

*exception* parser. **ParserError**

Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in **SyntaxError** raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to `sequence2st()` and an explanatory string. Calls to `sequence2st()` need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions `compilest()`, `expr()`, and `suite()` may raise exceptions which are normally raised by the parsing and compilation process. These include the built-in exceptions **MemoryError**, **OverflowError**, **SyntaxError**, and **SystemError**. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.
ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the pickle module) is also supported.

parser.STType

The type of the objects returned by expr(), suite() and sequence2st().

ST objects have the following methods:

ST.compile(filename='<syntax-tree>')

Same as compilest(st, filename).

ST.isexpr()

Same as isexpr(st).

ST.issuite()

Same as issuite(st).

ST.tolist(line_info=False, col_info=False)

Same as st2list(st, line_info, col_info).

ST.totuple(line_info=False, col_info=False)

Same as st2tuple(st, line_info, col_info).
Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the `parser` module to produce an intermediate data structure is equivalent to the code

```python
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the `parser` module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```python
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```python
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
```
return st, st.compile()
The `ast` module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing `ast.PyCF_ONLY_AST` as a flag to the `compile()` built-in function, or using the `parse()` helper provided in this module. The result will be a tree of objects whose classes all inherit from `ast.AST`. An abstract syntax tree can be compiled into a Python code object using the built-in `compile()` function.
Node classes

class ast.AST

This is the base of all AST node classes. The actual node classes are derived from the Parser/Python.asdl file, which is reproduced below. They are defined in the _ast C module and re-exported in ast.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, ast.stmt or ast.expr). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, ast.BinOp inherits from ast.expr. For production rules with alternatives (aka “sums”), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields

Each concrete class has an attribute _fields which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, ast.BinOp instances have an attribute left of type ast.expr.

If these attributes are marked as optional in the grammar (using a question mark), the value might be None. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with
compile().

**lineno**  
**col_offset**

Instances of `ast.expr` and `ast.stmt` subclasses have **lineno** and **col_offset** attributes. The **lineno** is the line number of source text (1-indexed so the first line is line 1) and the **col_offset** is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```python
node = ast.UnaryOp()  
node.op = ast.USub()  
node.operand = ast.Num()  
node.operand.n = 5  
node.operand.linenon = 0  
node.operand.col_offset = 0  
node.linenon = 0  
node.col_offset = 0
```

or the more compact

```python
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0))
```
Abstract Grammar

The abstract grammar is currently defined as follows:

```
module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

    stmt = FunctionDef(identifier name, arguments args
                        stmt* body, expr* decorator_list)
        | AsyncFunctionDef(identifier name, arguments args
                            stmt* body, expr* decorator_list)
        | ClassDef(identifier name,
                        expr* bases,
                        keyword* keywords,
                        stmt* body,
                        expr* decorator_list)
        | Return(expr? value)

    | Delete(expr* targets)
    | Assign(expr* targets, expr value)
    | AugAssign(expr target, operator op, expr value)

    -- not really an actual node but useful in Jython's typesystem.
    | Suite(stmt* body)

    expr = Call(function expr, expr* args)
        | Getattr(expr target, expr attr)
        | Getitem(expr target, expr index)
        | Getslice(expr target, expr? low, expr? high, expr? step)
        | Setattr(expr target, expr attr, expr value)
        | Setitem(expr target, expr index, expr value)
        | Setslice(expr target, expr? low, expr? high, expr? step, expr value)

    -- ASDL's 7 builtin types are:
    -- identifier, int, string, bytes, object, singleton
    -- singleton: None, True or False
    -- constant can be None, whereas None means "no value"
    -- 'simple' indicates that we annotate simple name without parens
```
AnnAssign(expr target, expr annotation, 
-- use 'orelse' because else is a keyword
| For(expr target, expr iter, stmt* body, 
| AsyncFor(expr target, expr iter, stmt* body, 
| While(expr test, stmt* body, stmt* orelse) 
| If(expr test, stmt* body, stmt* orelse) 
| With(withitem* items, stmt* body) 
| AsyncWith(withitem* items, stmt* body) 

| Raise(expr? exc, expr? cause) 
| Try(stmt* body, excepthandler* handlers, 
| Assert(expr test, expr? msg) 

| Import(alias* names) 
| ImportFrom(identifier? module, alias* names) 

| Global(identifier* names) 
| Nonlocal(identifier* names) 
| Expr(expr value) 
| Pass | Break | Continue 

-- XXX Jython will be different 
-- col_offset is the byte offset in the utf8 string the parser uses 
attributes (int lineno, int col_offset) 

-- BoolOp() can use left & right? 
expr = BoolOp(boolop op, expr* values) 
| BinOp(expr left, operator op, expr right) 
| UnaryOp(unaryop op, expr operand) 
| Lambda(arguments args, expr body) 
| IfExp(expr test, expr body, expr orelse) 
| Dict(expr* keys, expr* values) 
| Set(expr* elts) 
| ListComp(expr elt, comprehension* generators) 
| SetComp(expr elt, comprehension* generators) 
| DictComp(expr key, expr value, comprehension* generators)
GeneratorExp(expr elt, comprehension* generators)
-- the grammar constrains where yield expressions can occur
Await(expr value)
Yield(expr? value)
YieldFrom(expr value)
-- need sequences for compare to distinguish
-- x < 4 < 3 and (x < 4) < 3
Compare(expr left, cmpop* ops, expr* comparators)
Call(expr func, expr* args, keyword* keywords)
Num(object n) -- a number as a PyObject.
Str(string s) -- need to specify raw, unicode, etc?
FormattedValue(expr value, int? conversion, expr? format_spec)
JoinedStr(expr* values)
Bytes(bytes s)
NameConstant(singleton value)
Ellipsis
Constant(constant value)
-- the following expression can appear in an assignment
Attribute(expr value, identifier attr, expr_context ctx)
Subscript(expr value, slice slice, expr_context ctx)
Starred(expr value, expr_context ctx)
Name(identifier id, expr_context ctx)
List(expr* elts, expr_context ctx)
Tuple(expr* elts, expr_context ctx)

-- col_offset is the byte offset in the utf8 string the parser uses
attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore
slice = Slice(expr? lower, expr? upper, expr? step)
| ExtSlice(slice* dims)
| Index(expr value)

boolop = And | Or
operator = Add | Sub | Mult | MatMult | Div | Mod | RShift | BitOr | BitXor | BitAnd

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | Lte | Gt | Gte | Is |

comprehension = (expr target, expr iter, expr* ifs)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)

arguments = (arg* args, arg? vararg, arg* kwonly, arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation)

-- keyword arguments supplied to call (NULL identifier for **kwargs)
keyword = (identifier? arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)

withitem = (expr context_expr, expr? optional_vars)
ast Helpers

Apart from the node classes, the ast module defines these utility functions and classes for traversing abstract syntax trees:

**ast.parse**(source, filename='unknown', mode='exec')

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

**Warning**: It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python’s AST compiler.

**ast.literal_eval**(node_or_string)

Safely evaluate an expression node or a string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and `None`.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

**Warning**: It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python’s AST compiler.

*Changed in version 3.2*: Now allows bytes and set literals.

**ast.get_docstring**(node, clean=True)
Return the docstring of the given node (which must be a FunctionDef, AsyncFunctionDef, ClassDef, or Module node), or None if it has no docstring. If clean is true, clean up the docstring’s indentation with inspect.cleandoc().

Changed in version 3.5: AsyncFunctionDef is now supported.

ast.fix_missing_locations(node)
When you compile a node tree with compile(), the compiler expects lineno and col_offset attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at node.

ast.increment_lineno(node, n=1)
Increment the line number of each node in the tree starting at node by n. This is useful to “move code” to a different location in a file.

ast.copy_location(new_node, old_node)
Copy source location (lineno and col_offset) from old_node to new_node if possible, and return new_node.

ast.iter_fields(node)
Yield a tuple of (fieldname, value) for each field in node._fields that is present on node.

ast.iter_child_nodes(node)
Yield all direct child nodes of node, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.walk(node)
Recursively yield all descendant nodes in the tree starting at node (including node itself), in no specified order. This is useful if you only want to modify nodes in place and don’t care about the context.

class ast.NodeVisitor
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the visit() method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(node)
Visit a node. The default implementation calls the method called self.visit_classname where classname is the name of the node class, or generic_visit() if that method doesn’t exist.

generic_visit(node)
This visitor calls visit() on all children of the node.

Note that child nodes of nodes that have a custom visitor method won’t be visited unless the visitor calls generic_visit() or visits them itself.

Don’t use the NodeVisitor if you want to apply changes to nodes during traversal. For this a special visitor exists (NodeTransformer) that allows modifications.

class ast.NodeTransformer
A NodeVisitor subclass that walks the abstract syntax tree and allows modification of nodes.
The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```python
class RewriteName(NodeTransformer):
    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```python
node = YourTransformer().visit(node)
```

```python
ast.dump(node, annotate_fields=True, include_attributes=False)
```

Return a formatted dump of the tree in `node`. This is mainly useful for debugging purposes. The returned string will show the names
and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to `True`.

**See also:** *Green Tree Snakes*, an external documentation resource, has good details on working with Python ASTs.
**symtable** — Access to the compiler’s symbol tables

**Source code:** Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. **symtable** provides an interface to examine these tables.
Generating Symbol Tables

`symtable.symtable(code, filename, compile_type)`

Return the toplevel `SymbolTable` for the Python source `code`. `filename` is the name of the file containing the code. `compile_type` is like the `mode` argument to `compile()`.
Examining Symbol Tables

class symtable.SymbolTable
    A namespace table for a block. The constructor is not public.

    get_type()
        Return the type of the symbol table. Possible values are 'class', 'module', and 'function'.

    get_id()
        Return the table's identifier.

    get_name()
        Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or 'top' if the table is global (get_type() returns 'module').

    get_lineno()
        Return the number of the first line in the block this table represents.

    is_optimized()
        Return True if the locals in this table can be optimized.

    is_nested()
        Return True if the block is a nested class or function.

    has_children()
        Return True if the block has nested namespaces within it. These can be obtained with get_children().
has_exec()  
Return True if the block uses exec.

get_identifiers()  
Return a list of names of symbols in this table.

lookup(name)  
Lookup name in the table and return a Symbol instance.

get_symbols()  
Return a list of Symbol instances for names in the table.

get_children()  
Return a list of the nested symbol tables.

class symtable.Function  
A namespace for a function or method. This class inherits SymbolTable.

get_parameters()  
Return a tuple containing names of parameters to this function.

get_locals()  
Return a tuple containing names of locals in this function.

get Globals()  
Return a tuple containing names of globals in this function.

get_frees()  
Return a tuple containing names of free variables in this function.

class symtable.Class
A namespace of a class. This class inherits `SymbolTable`.

```python
get_methods()
Return a tuple containing the names of methods declared in the class.
```

class symtable.Symbol
An entry in a `SymbolTable` corresponding to an identifier in the source. The constructor is not public.

```python
get_name()
Return the symbol's name.
```

```python
is_referenced()
Return `True` if the symbol is used in its block.
```

```python
is_imported()
Return `True` if the symbol is created from an import statement.
```

```python
is_parameter()
Return `True` if the symbol is a parameter.
```

```python
is_global()
Return `True` if the symbol is global.
```

```python
isDeclaredGlobal()
Return `True` if the symbol is declared global with a global statement.
```

```python
is_local()
Return `True` if the symbol is local to its block.
```

```python
is_free()
```
Return `True` if the symbol is referenced in its block, but not assigned to.

**is_assigned()**

Return `True` if the symbol is assigned to in its block.

**is_namespace()**

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```python
global table

>>> table = symtable.symtable("def some_func(): pass")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

**get_namespaces()**

Return a list of namespaces bound to this name.

**get_namespace()**

Return the namespace bound to this name. If more than one namespace is bound, `ValueError` is raised.
symbol — Constants used with Python parse trees

Source code: Lib/symbol.py

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:

symbol.sym_name

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.
**token — Constants used with Python parse trees**

**Source code:** [Lib/token.py](https://www.python.org/)

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Grammar` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

**token**. `tok_name`

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

**token**. `ISTERMINAL(x)`

Return true for terminal token values.

**token**. `ISNONTERMINAL(x)`

Return true for non-terminal token values.

**token**. `ISEOF(x)`

Return true if x is the marker indicating the end of input.
The token constants are:

token. **ENDMARKER**
token. **NAME**
token. **NUMBER**
token. **STRING**
token. **NEWLINE**
token. **INDENT**
token. **DEDENT**
token. **LPAR**
token. **RPAR**
token. **LSQB**
token. **RSQB**
token. **COLON**
token. **COMMA**
token. **SEMI**
token. **PLUS**
token. **MINUS**
token. **STAR**
token. **SLASH**
token. **VBAR**
token. **AMPER**
token. **LESS**
token. **GREATER**
token. **EQUAL**
token. **DOT**
token. **PERCENT**
token. **LBRACE**
token. **RBRACE**
token. **EQEQUAL**
token. **NOTEQUAL**
token. **LESSEQUAL**
token. **GREATEREQUAL**
The following token type values aren’t used by the C tokenizer but are needed for the `tokenize` module.

**token.COMMENT**

Token value used to indicate a comment.
token. \texttt{NL}

Token value used to indicate a non-terminating newline. The \texttt{NEWLINE} token indicates the end of a logical line of Python code; \texttt{NL} tokens are generated when a logical line of code is continued over multiple physical lines.

token. \texttt{ENCODING}

Token value that indicates the encoding used to decode the source bytes into text. The first token returned by \texttt{tokenize.tokenize()} will always be an \texttt{ENCODING} token.

\textit{Changed in version 3.5}: Added \texttt{AWAIT} and \texttt{ASYNC} tokens.

\textit{Changed in version 3.7}: Added \texttt{COMMENT}, \texttt{NL} and \texttt{ENCODING} tokens.

\textit{Changed in version 3.7}: Removed \texttt{AWAIT} and \texttt{ASYNC} tokens. “async” and “await” are now tokenized as \texttt{NAME} tokens.
**keyword** — Testing for Python keywords

**Source code:** `Lib/keyword.py`

This module allows a Python program to determine if a string is a keyword.

`keyword.iskeyword(s)`

Return true if `s` is a Python keyword.

`keyword.kwlist`

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.
The `tokenize` module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing “pretty-printers,” including colorizers for on-screen displays.

To simplify token stream handling, all `operator` and `delimiter` tokens and `Ellipsis` are returned using the generic `OP` token type. The exact type can be determined by checking the `exact_type` property on the `named tuple` returned from `tokenize.tokenize()`.
Tokenizing Input

The primary entry point is a generator:

tokenize. **`tokenize`**(readline)

The `tokenize()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `io.IOBase.readline()` method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included. The 5 tuple is returned as a named tuple with the field names: `type` `string` `start` `end` `line`.

The returned named tuple has an additional property named `exact_type` that contains the exact operator type for **OP** tokens. For all other token types `exact_type` equals the named tuple `type` field.

*Changed in version 3.1:* Added support for named tuples.

*Changed in version 3.3:* Added support for `exact_type`.

tokenize() determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

All constants from the `token` module are also exported from
Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

**tokenize.**\texttt{untokenize(iterable)}

Converts tokens back into Python source code. The \textit{iterable} must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the \texttt{ENCODING} token, which is the first token sequence output by \texttt{tokenize()}.  

**tokenize()** needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

**tokenize.**\texttt{detect\_encoding(readline)}

The \texttt{detect\_encoding()} function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, \texttt{readline}, in the same way as the \texttt{tokenize()} generator.

It will call \texttt{readline} a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.
It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in PEP 263. If both a BOM and a cookie are present, but disagree, a SyntaxError will be raised. Note that if the BOM is found, 'utf-8-sig' will be returned as an encoding.

If no encoding is specified, then the default of 'utf-8' will be returned.

Use open() to open Python source files: it uses detect_encoding() to detect the file encoding.

tokenize.open(filename)
Open a file in read only mode using the encoding detected by detect_encoding().

New in version 3.2.

exception tokenize.TokenError
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

"""Beginning of
docstring

or:

[1,
  2,
  3

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as ERRORTOKEN, followed by the
tokenization of their contents.
Command-Line Usage

New in version 3.3.

The `tokenize` module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

- **-h, --help**
  show this help message and exit

- **-e, --exact**
  display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.
Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```python
from tokenize import tokenize, untokenize, NUMBER, STRING
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements."

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library. Known cases are "e-007" (Windows) and "e-07" (not Windows). Since we're only showing 12 digits, and the 13th isn't close to 5, the rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    ""

    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([toknum, tokval])
```

```
Example of tokenizing from the command line. The script:

```python
def say_hello():
    print("Hello, World!")
say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)
The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
```
tabnanny — Detection of ambiguous indentation

Source code: Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function `check()` described below.

**Note:** The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

`tabnanny.check(file_or_dir)`

If `file_or_dir` is a directory and not a symbolic link, then recursively descend the directory tree named by `file_or_dir`, checking all `.py` files along the way. If `file_or_dir` is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the `print()` function.

`tabnanny.verbose`

Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

`tabnanny.filename_only`

Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.
exception tabnanny. **NannyNag**

Raised by `process_tokens()` if detecting an ambiguous indent. Captured and handled in `check()`.

```python
import tabnanny

tabnanny.process_tokens(tokens)
```

This function is used by `check()` to process tokens generated by the `tokenize` module.

**See also:**

**Module tokenize**

Lexical scanner for Python source code.
pyclbr — Python class browser support

Source code: Lib/pyclbr.py

The `pyclbr` module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

`pyclbr.readmodule(module, path=None)`

Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter `module` is a string with the name of the module to read; it may be the name of a module within a package. If given, `path` is a sequence of directory paths prepended to `sys.path`, which is used to locate the module source code.

`pyclbr.readmodule_ex(module, path=None)`

Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a `def` or `class` statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with `readmodule`, `module` names the module to be read and `path` is
prepended to sys.path. If the module being read is a package, the returned dictionary has a key '__path__' whose value is a list containing the package search path.

New in version 3.7: Descriptors for nested definitions. They are accessed through the new children attribute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.
Function Objects

Class Function instances describe functions defined by def statements. They have the following attributes:

Function. **file**
   Name of the file in which the function is defined.

Function. **module**
   The name of the module defining the function described.

Function. **name**
   The name of the function.

Function. **lineno**
   The line number in the file where the definition starts.

Function. **parent**
   For top-level functions, None. For nested functions, the parent.

   *New in version 3.7.*

Function. **children**
   A dictionary mapping names to descriptors for nested functions and classes.

   *New in version 3.7.*
Class Objects

Class Class instances describe classes defined by class statements. They have the same attributes as Functions and two more.

Class. **file**
   Name of the file in which the class is defined.

Class. **module**
   The name of the module defining the class described.

Class. **name**
   The name of the class.

Class. **lineno**
   The line number in the file where the definition starts.

Class. **parent**
   For top-level classes, None. For nested classes, the parent.
   
   *New in version 3.7.*

Class. **children**
   A dictionary mapping names to descriptors for nested functions and classes.

   *New in version 3.7.*

Class. **super**
   A list of Class objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by *readmodule_ex()* are listed as a string with the class name.
instead of as Class objects.

Class. **methods**

A dictionary mapping method names to line numbers. This can be derived from the newer children dictionary, but remains for back-compatibility.
**py_compile** — Compile Python source files

**Source code:** Lib/py_compile.py

The `py_compile` module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

**exception py_compile.PyCompileError**

Exception raised when an error occurs while attempting to compile the file.

`py_compile.compile(file, cfile=None, dfile=None, doraise=False, optimize=-1, invalidation_mode=PycInvalidationMode.TIMESTAMP)`

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named `file`. The byte-code is written to `cfile`, which defaults to the PEP 3147/PEP 488 path, ending in `.pyc`. For example, if `file` is `/foo/bar/baz.py` then `cfile` will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If `dfile` is specified, it is used as the name of the source file in error messages when instead of `file`. If `doraise` is true, a `PyCompileError` is raised when an error is encountered while
compiling file. If doraise is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever `cfile` value was used.

If the path that `cfile` becomes (either explicitly specified or computed) is a symlink or non-regular file, `FileExistsError` will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

`optimize` controls the optimization level and is passed to the built-in `compile()` function. The default of `-1` selects the optimization level of the current interpreter.

`invalidation_mode` should be a member of the `PycInvalidationMode` enum and controls how the generated bytecode cache is invalidated at runtime. The default is `PycInvalidationMode.CHECKED_HASH` if the `SOURCE_DATE_EPOCH` environment variable is set, otherwise the default is `PycInvalidationMode.TIMESTAMP`.

*Changed in version 3.2:* Changed default value of `cfile` to be PEP 3147-compliant. Previous default was `file + 'c'` (`'o'` if optimization was enabled). Also added the `optimize` parameter.

*Changed in version 3.4:* Changed code to use `importlib` for the byte-code cache file writing. This means file creation/writing semantics now match what `importlib` does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that
FileExistsError is raised if `cfile` is a symlink or non-regular file.

*Changed in version 3.7:* The `invalidation_mode` parameter was added as specified in PEP 552. If the `SOURCE_DATE_EPOCH` environment variable is set, `invalidation_mode` will be forced to `PycInvalidationMode.CHECKED_HASH`.

*Changed in version 3.7.2:* The `SOURCE_DATE_EPOCH` environment variable no longer overrides the value of the `invalidation_mode` argument, and determines its default value instead.

```python
class py_compile.PycInvalidationMode
```
A enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See Cached bytecode invalidation for more information on how Python invalidates `.pyc` files at runtime.

*New in version 3.7.*

**TIMESTAMP**

The `.pyc` file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the `.pyc` file needs to be regenerated.

**CHECKED_HASH**

The `.pyc` file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the `.pyc` file needs to be regenerated.

**UNCHECKED_HASH**

Like `CHECKED_HASH`, the `.pyc` file includes a hash of the
source file content. However, Python will at runtime assume the `.pyc` file is up to date and not validate the `.pyc` against the source file at all.

This option is useful when the `.pycs` are kept up to date by some system external to Python like a build system.

py_compile. `main(args=None)`

Compile several source files. The files named in `args` (or on the command line, if `args` is `None`) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If `'-'` is the only parameter in `args`, the list of files is taken from standard input.

*Changed in version 3.2:* Added support for `'-'`.

When this module is run as a script, the `main()` is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

**See also:**

**Module** `compileall`

Utilities to compile all Python source files in a directory tree.
compileall — Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.
Command-line use

This module can work as a script (using `python -m compileall`) to compile Python sources.

**directory ...**
**file ...**

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

- `l`
  Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

- `f`
  Force rebuild even if timestamps are up-to-date.

- `q`
  Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (`-qq`), all output is suppressed.

- `d destdir`
  Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled into the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

- `x regex`
regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

- **i** list
  Read the file list and add each line that it contains to the list of files and directories to compile. If list is -, read lines from stdin.

- **b**
  Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their PEP 3147 locations and names, which allows byte-code files from multiple versions of Python to coexist.

- **r**
  Control the maximum recursion level for subdirectories. If this is given, then -l option will not be taken into account. python -m compileall <directory> -r 0 is equivalent to python -m compileall <directory> -l.

- **j** N
  Use N workers to compile the files within the given directory. If 0 is used, then the result of os.cpu_count() will be used.

- **--invalidation-mode** [timestamp|checked-hash|unchecked-hash]
  Control how the generated byte-code files are invalidated at runtime. The timestamp value, means that .pyc files with the source timestamp and size embedded will be generated. The checked-hash and unchecked-hash values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the
source file contents rather than a timestamp. See **Cached bytecode invalidation** for more information on how Python validates bytecode cache files at runtime. The default is `timestamp` if the `SOURCE_DATE_EPOCH` environment variable is not set, and `checked-hash` if the `SOURCE_DATE_EPOCH` environment variable is set.

*Changed in version 3.2:* Added the `-i`, `-b` and `-h` options.

*Changed in version 3.5:* Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

*Changed in version 3.7:* Added the `--invalidation-mode` option.

There is no command-line option to control the optimization level used by the `compile()` function, because the Python interpreter itself already provides the option: `python -O -m compileall`. 
Public functions

compileall.compile_dir(dir, maxlevels=10, ddir=None, force=False, rx=None, quiet=0, legacy=False, optimize=-1, workers=1, invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP)

Recursively descend the directory tree named by dir, compiling all .py files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The maxlevels parameter is used to limit the depth of the recursion; it defaults to 10.

If ddir is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If force is true, modules are re-compiled even if the timestamps are up to date.

If rx is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

If quiet is False or 0 (the default), the filenames and other information are printed to standard out. Set to 1, only errors are printed. Set to 2, all output is suppressed.

If legacy is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their PEP
locations and names, which allows byte-code files from multiple versions of Python to coexist.

`optimize` specifies the optimization level for the compiler. It is passed to the built-in `compile()` function.

The argument `workers` specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can’t use multiple workers and `workers` argument is given, then sequential compilation will be used as a fallback. If `workers` is lower than 0, a `ValueError` will be raised.

`invalidation_mode` should be a member of the `py_compile.PycInvalidationMode` enum and controls how the generated pycs are invalidated at runtime.

*Changed in version 3.2:* Added the `legacy` and `optimize` parameter.

*Changed in version 3.5:* Added the `workers` parameter.

*Changed in version 3.5:* The `quiet` parameter was changed to a multilevel value.

*Changed in version 3.5:* The `legacy` parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of `optimize` is.

*Changed in version 3.6:* Accepts a `path-like object`.

*Changed in version 3.7:* The `invalidation_mode` parameter was added.

```python
compileall.compile_file(fullname, ddir=None, force=False, rx=None, quiet=0, legacy=False, optimize=-1, invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP)
```
Compile the file with path fullname. Return a true value if the file compiled successfully, and a false value otherwise.

If ddir is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If rx is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and True is returned.

If quiet is False or 0 (the default), the filenames and other information are printed to standard out. Set to 1, only errors are printed. Set to 2, all output is suppressed.

If legacy is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their PEP 3147 locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in compile() function.

invalidation_mode should be a member of the py_compile.PycInvalidationMode enum and controls how the generated pycs are invalidated at runtime.

New in version 3.2.

Changed in version 3.5: quiet parameter was changed to a multilevel value.
Changed in version 3.5: The legacy parameter only writes out .pyc files, not .pyo files no matter what the value of optimize is.

Changed in version 3.7: The invalidation_mode parameter was added.

```
compileall.compile_path(skip_curdir=True, maxlevels=0, force=False, quiet=0, legacy=False, optimize=-1, invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP)
```

Byte-compile all the .py files found along sys.path. Return a true value if all the files compiled successfully, and a false value otherwise.

If skip_curdir is true (the default), the current directory is not included in the search. All other parameters are passed to the compile_dir() function. Note that unlike the other compile functions, maxlevels defaults to 0.

Changed in version 3.2: Added the legacy and optimize parameter.

Changed in version 3.5: quiet parameter was changed to a multilevel value.

Changed in version 3.5: The legacy parameter only writes out .pyc files, not .pyo files no matter what the value of optimize is.

Changed in version 3.7: The invalidation_mode parameter was added.

To force a recompile of all the .py files in the Lib/ subdirectory and all its subdirectories:

```
import compileall
```
```
compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\]\.svn'))

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

**See also:**

**Module py_compile**

Byte-compile a single source file.
**dis — Disassembler for Python bytecode**

**Source code:** Lib/dis.py

The `dis` module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

**CPython implementation detail:** Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

*Changed in version 3.6:* Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Example: Given the function `myfunc()`:

```python
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```python
>>> dis.dis(myfunc)
  2  0 LOAD_GLOBAL           0 (len)
  2  2 LOAD_FAST             0 (alist)
  4  4 CALL_FUNCTION         1
```
The “2” is a line number.

6 RETURN_VALUE
Bytecode analysis

*New in version 3.4.*

The bytecode analysis API allows pieces of Python code to be wrapped in a `Bytecode` object that provides easy access to details of the compiled code.

```python
class dis.Bytecode(x, *, first_line=None, current_offset=None)
```

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by `compile()`).

This is a convenience wrapper around many of the functions listed below, most notably `get_instructions()`, as iterating over a `Bytecode` instance yields the bytecode operations as `Instruction` instances.

If `first_line` is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If `current_offset` is not `None`, it refers to an instruction offset in the disassembled code. Setting this means `dis()` will display a “current instruction” marker against the specified opcode.

```python
classmethod from_traceback(tb)
```

Construct a `Bytecode` instance from the given traceback, setting `current_offset` to the instruction responsible for the exception.
**codeobj**

The compiled code object.

**first_line**

The first source line of the code object (if available)

**dis()**

Return a formatted view of the bytecode operations (the same as printed by `dis.dis()`, but returned as a multi-line string).

**info()**

Return a formatted multi-line string with detailed information about the code object, like `code_info()`.

*Changed in version 3.7:* This can now handle coroutine and asynchronous generator objects.

Example:

```python
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...    print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```
Analysis functions

The `dis` module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn’t useful:

`dis.code_info(x)`

Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

*New in version 3.2.*

*Changed in version 3.7:* This can now handle coroutine and asynchronous generator objects.

`dis.show_code(x, *, file=None)`

Print detailed code object information for the supplied function, method, source code string or code object to `file` (or `sys.stdout` if `file` is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

*New in version 3.2.*

*Changed in version 3.4:* Added `file` parameter.
dis.dis(x=None, *, file=None, depth=None)
Disassemble the x object. x can denote either a module, a class, a
method, a function, a generator, an asynchronous generator, a
coroutine, a code object, a string of source code or a byte
sequence of raw bytecode. For a module, it disassembles all
functions. For a class, it disassembles all methods (including class
and static methods). For a code object or sequence of raw
bytecode, it prints one line per bytecode instruction. It also
recursively disassembles nested code objects (the code of
comprehensions, generator expressions and nested functions, and
the code used for building nested classes). Strings are first
compiled to code objects with the compile() built-in function
before being disassembled. If no object is provided, this function
disassembles the last traceback.

The disassembly is written as text to the supplied file argument if
provided and to sys.stdout otherwise.

The maximal depth of recursion is limited by depth unless it is
None. depth=0 means no recursion.

*Changed in version 3.4:* Added file parameter.

*Changed in version 3.7:* Implemented recursive disassembling and
added depth parameter.

*Changed in version 3.7:* This can now handle coroutine and
asynchronous generator objects.

dis.distb(tb=None, *, file=None)
Disassemble the top-of-stack function of a traceback, using the last
traceback if none was passed. The instruction causing the
exception is indicated.
The disassembly is written as text to the supplied `file` argument if provided and to `sys.stdout` otherwise.

*Changed in version 3.4:* Added `file` parameter.

```python
dis.disassemble(code, lasti=-1, *, file=None)
```
```python
dis.disco(code, lasti=-1, *, file=None)
```

Disassemble a code object, indicating the last instruction if `lasti` was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`
3. a labelled instruction, indicated with `>>`
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied `file` argument if provided and to `sys.stdout` otherwise.

*Changed in version 3.4:* Added `file` parameter.

```python
dis.get_instructions(x, *, first_line=None)
```

Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of `Instruction` named tuples giving the details of each operation in the supplied code.

If `first_line` is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code.
Otherwise, the source line information (if any) is taken directly from the disassembled code object.

*New in version 3.4.*

**dis.** `findlinestarts(code)`

This generator function uses the `co_firstlinenno` and `co_lnotab` attributes of the code object `code` to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs. See `Objects/Lnotab_notes.txt` for the `co_lnotab` format and how to decode it.

*Changed in version 3.6:* Line numbers can be decreasing. Before, they were always increasing.

**dis.** `findlabels(code)`

Detect all offsets in the code object `code` which are jump targets, and return a list of these offsets.

**dis.** `stack_effect(opcode[, oparg])`

Compute the stack effect of `opcode` with argument `oparg`.

*New in version 3.4.*
Python Bytecode Instructions

The `get_instructions()` function and `Bytecode` class provide details of bytecode instructions as `Instruction` instances:

```python
class dis.Instruction
    Details for a bytecode operation

    opcode
        numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the Opcode collections.

    opname
        human readable name for operation

    arg
        numeric argument to operation (if any), otherwise None

    argval
        resolved arg value (if known), otherwise same as arg

    argrepr
        human readable description of operation argument

    offset
        start index of operation within bytecode sequence

    starts_line
        line started by this opcode (if any), otherwise None

    is_jump_target
        True if other code jumps to here, otherwise False
```
New in version 3.4.

The Python compiler currently generates the following bytecode instructions.

**General instructions**

**NOP**
Do nothing code. Used as a placeholder by the bytecode optimizer.

**POP_TOP**
Removes the top-of-stack (TOS) item.

**ROT_TWO**
Swaps the two top-most stack items.

**ROT_THREE**
Lifts second and third stack item one position up, moves top down to position three.

**DUP_TOP**
Duplicates the reference on top of the stack.

New in version 3.2.

**DUP_TOP_TWO**
Duplicates the two references on top of the stack, leaving them in the same order.

New in version 3.2.

**Unary operations**

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.
UNARY_POSITIVE
  Implements $TOS = +TOS$.

UNARY_NEGATIVE
  Implements $TOS = -TOS$.

UNARY_NOT
  Implements $TOS = \text{not } TOS$.

UNARY_INVERT
  Implements $TOS = \sim TOS$.

GET_ITER
  Implements $TOS = \text{iter}(TOS)$.

GET_YIELD_FROM_ITER
  If $TOS$ is a generator iterator or coroutine object it is left as is. Otherwise, implements $TOS = \text{iter}(TOS)$.

  New in version 3.5.

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
  Implements $TOS = TOS1 \times TOS$.

BINARY_MULTIPLY
  Implements $TOS = TOS1 \times TOS$.

BINARY_MATRIX_MULTIPLY
Implements \( TOS = TOS1 \oplus TOS \).

*New in version 3.5.*

**BINARY_FLOOR_DIVIDE**

Implements \( TOS = TOS1 \div TOS \).

**BINARY_TRUE_DIVIDE**

Implements \( TOS = TOS1 \div TOS \).

**BINARY_MODULO**

Implements \( TOS = TOS1 \mod TOS \).

**BINARY_ADD**

Implements \( TOS = TOS1 + TOS \).

**BINARY_SUBTRACT**

Implements \( TOS = TOS1 - TOS \).

**BINARY_SUBSCR**

Implements \( TOS = TOS1[TOS] \).

**BINARY_LSHIFT**

Implements \( TOS = TOS1 << TOS \).

**BINARY_RSHIFT**

Implements \( TOS = TOS1 >> TOS \).

**BINARY_AND**

Implements \( TOS = TOS1 \& TOS \).

**BINARY_XOR**

Implements \( TOS = TOS1 \oplus TOS \).
**BINARY_OR**

Implements $TOS = TOS1 \mid TOS$.

**In-place operations**

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

**INPLACE_POWER**

Implements in-place $TOS = TOS1 \, ** \, TOS$.

**INPLACE_MULTIPLY**

Implements in-place $TOS = TOS1 \, * \, TOS$.

**INPLACE_MATRIX_MULTIPLY**

Implements in-place $TOS = TOS1 \, @ \, TOS$.

*New in version 3.5.*

**INPLACE_FLOOR_DIVIDE**

Implements in-place $TOS = TOS1 \, // \, TOS$.

**INPLACE_TRUE_DIVIDE**

Implements in-place $TOS = TOS1 \, / \, TOS$.

**INPLACE_MODULO**

Implements in-place $TOS = TOS1 \, \% \, TOS$.

**INPLACE_ADD**

Implements in-place $TOS = TOS1 \, + \, TOS$.

**INPLACE_SUBTRACT**
Implements in-place $TOS = TOS1 - TOS$.

**INPLACE_LSHIFT**
Implements in-place $TOS = TOS1 \ll TOS$.

**INPLACE_RSHIFT**
Implements in-place $TOS = TOS1 \gg TOS$.

**INPLACE_AND**
Implements in-place $TOS = TOS1 \& TOS$.

**INPLACE_XOR**
Implements in-place $TOS = TOS1 \^ TOS$.

**INPLACE_OR**
Implements in-place $TOS = TOS1 \mid TOS$.

**STORE_SUBSCR**
Implements $TOS1[TOS] = TOS2$.

**DELETE_SUBSCR**
Implements `del TOS1[TOS]`.

Coroutine opcodes

**GET_AWAITABLE**
Implements $TOS = \text{get\_awaitable}(TOS)$, where \text{get\_awaitable}(o) returns o if o is a coroutine object or a generator object with the CO_ITERABLE_COROUTINE flag, or resolves o.__await__.

*New in version 3.5.*
GET_AITER
Implements TOS = TOS.__aiter__().

*New in version 3.5.*

*Changed in version 3.7:* Returning awaitable objects from __aiter__ is no longer supported.

GET_ANEXT
Implements PUSH(get_awaitable(TOS.__anext__())). See GET_AWAITABLE for details about get_awaitable

*New in version 3.5.*

BEFORE_ASYNC_WITH
Resolves __aenter__ and __aexit__ from the object on top of the stack. Pushes __aexit__ and result of __aenter__() to the stack.

*New in version 3.5.*

SETUP_ASYNC_WITH
Creates a new frame object.

*New in version 3.5.*

Miscellaneous opcodes

PRINT_EXPR
Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with POP_TOP.

BREAK_LOOP
Terminates a loop due to a `break` statement.

**CONTINUE_LOOP**(target)
Continues a loop due to a `continue` statement. `target` is the address to jump to (which should be a `FOR_ITER` instruction).

**SET_ADD**(i)
Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

**LIST_APPEND**(i)
Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

**MAP_ADD**(i)
Calls `dict.setitem(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.

*New in version 3.1.*

For all of the `SET_ADD`, `LIST_APPEND` and `MAP_ADD` instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

**RETURN_VALUE**
Returns with TOS to the caller of the function.

**YIELD_VALUE**
Pops TOS and yields it from a `generator`.

**YIELD_FROM**
Pops TOS and delegates to it as a subiterator from a `generator`. 
New in version 3.3.

**SETUP_ANNOTATIONS**
Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains variable annotations statically.

New in version 3.6.

**IMPORT_STAR**
Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

**POP_BLOCK**
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

**POP_EXCEPT**
Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

**END_FINALLY**
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

**LOAD_BUILD_CLASS**
Pushes `builtins.__build_class__()` onto the stack. It is later called by `CALL_FUNCTION` to construct a class.
**SETUP_WITH**(delta)

This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by WITH_CLEANUP. Then, `__enter__()` is called, and a finally block pointing to `delta` is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (POP_TOP), or store it in (a) variable(s) (STORE_FAST, STORE_NAME, or UNPACK_SEQUENCE).

*New in version 3.2.*

**WITH_CLEANUP_START**

Cleans up the stack when a with statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

- SECOND = None
- (SECOND, THIRD) = (WHY_{RETURN, CONTINUE}), retval
- SECOND = WHY_*; no retval below it
- (SECOND, THIRD, FOURTH) = exc_info()

In the last case, TOS(SECOND, THIRD, FOURTH) is called, otherwise TOS(None, None, None). Pushes SECOND and result of the call to the stack.

**WITH_CLEANUP_FINISH**

Pops exception type and result of ‘exit’ function call from the stack.

If the stack represents an exception, and the function call returns a ‘true’ value, this information is “zapped” and replaced with a single WHY_SILENCED to prevent END_FINALLY from re-raising the exception. (But non-local gotos will still be resumed.)
All of the following opcodes use their arguments.

**STORE_NAME**(namei)
Implements name = TOS. namei is the index of name in the attribute co_names of the code object. The compiler tries to use **STORE_FAST** or **STORE_GLOBAL** if possible.

**DELETE_NAME**(namei)
Implements del name, where namei is the index into co_names attribute of the code object.

**UNPACK_SEQUENCE**(count)
Unpacks TOS into count individual values, which are put onto the stack right-to-left.

**UNPACK_EX**(counts)
Implements assignment with a starred target: Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The low byte of counts is the number of values before the list value, the high byte of counts the number of values after it. The resulting values are put onto the stack right-to-left.

**STORE_ATTR**(namei)
Implements TOS.name = TOS1, where namei is the index of name in co_names.

**DELETE_ATTR**(namei)
Implements del TOS.name, using namei as index into co_names.
**STORE_GLOBAL**(namei)
Works as **STORE_NAME**, but stores the name as a global.

**DELETE_GLOBAL**(namei)
Works as **DELETE_NAME**, but deletes a global name.

**LOAD_CONST**(consti)
Pushes co_consts[consti] onto the stack.

**LOAD_NAME**(namei)
Pushes the value associated with co_names[namei] onto the stack.

**BUILD_TUPLE**(count)
Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

**BUILD_LIST**(count)
Works as **BUILD_TUPLE**, but creates a list.

**BUILD_SET**(count)
Works as **BUILD_TUPLE**, but creates a set.

**BUILD_MAP**(count)
Pushes a new dictionary object onto the stack. Pops 2 * count items so that the dictionary holds count entries: {..., TOS3: TOS2, TOS1: TOS}.

*Changed in version 3.5:* The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold count items.

**BUILD_CONST_KEY_MAP**(count)
The version of BUILD_MAP specialized for constant keys. count values are consumed from the stack. The top element on the stack contains a tuple of keys.

New in version 3.6.

**BUILD_STRING**(count)
Concatenates count strings from the stack and pushes the resulting string onto the stack.

New in version 3.6.

**BUILD_TUPLE_UNPACK**(count)
Pops count iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays (*x, *y, *z).

New in version 3.5.

**BUILD_TUPLE_UNPACK_WITH_CALL**(count)
This is similar to BUILD_TUPLE_UNPACK, but is used for f(*x, *y, *z) call syntax. The stack item at position count + 1 should be the corresponding callable f.

New in version 3.6.

**BUILD_LIST_UNPACK**(count)
This is similar to BUILD_TUPLE_UNPACK, but pushes a list instead of tuple. Implements iterable unpacking in list displays [*x, *y, *z].

New in version 3.5.

**BUILD_SET_UNPACK**(count)
This is similar to `BUILD_TUPLE_UNPACK`, but pushes a set instead of tuple. Implements iterable unpacking in set displays \{\*x, \*y, \*z\}.

*New in version 3.5.*

**BUILD_MAP_UNPACK**(count)

Pops `count` mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays \{**x, **y, **z\}.

*New in version 3.5.*

**BUILD_MAP_UNPACK_WITH_CALL**(count)

This is similar to `BUILD_MAP_UNPACK`, but is used for \(f(**x, **y, **z)\) call syntax. The stack item at position `count + 2` should be the corresponding callable `f`.

*New in version 3.5.*

*Changed in version 3.6:* The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

**LOAD_ATTR**(namei)

Replaces TOS with `getattr(TOS, co_names[namei])`.

**COMPARE_OP**(opname)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

**IMPORT_NAME**(namei)

Imports the module `co_names[namei]`. TOS and TOS1 are
popped and provide the `fromlist` and `level` arguments of `__import__()`. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent `STORE_FAST` instruction modifies the namespace.

**IMPORT_FROM**(namei)

Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

**JUMP_FORWARD**(delta)

Increments bytecode counter by `delta`.

**POP_JUMP_IF_TRUE**(target)

If TOS is true, sets the bytecode counter to `target`. TOS is popped.

*New in version 3.1.*

**POP_JUMP_IF_FALSE**(target)

If TOS is false, sets the bytecode counter to `target`. TOS is popped.

*New in version 3.1.*

**JUMP_IF_TRUE_OR_POP**(target)

If TOS is true, sets the bytecode counter to `target` and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

*New in version 3.1.*

**JUMP_IF_FALSE_OR_POP**(target)

If TOS is false, sets the bytecode counter to `target` and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

*New in version 3.1.*
JUMP_ABSOLUTE(target)
Set bytecode counter to target.

FOR_ITER(delta)
TOS is an iterator. Call its __next__() method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by delta.

LOAD_GLOBAL(namei)
Loads the global named co_names[namei] onto the stack.

SETUP_LOOP(delta)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of delta bytes.

SETUP_EXCEPT(delta)
Pushes a try block from a try-except clause onto the block stack. delta points to the first except block.

SETUP_FINALLY(delta)
Pushes a try block from a try-except clause onto the block stack. delta points to the finally block.

LOAD_FAST(var_num)
Pushes a reference to the local co_varnames[var_num] onto the stack.

STORE_FAST(var_num)
Stores TOS into the local co_varnames[var_num].

DELETE_FAST(var_num)
Deletes local co_varnames[var_num].
**LOAD_CLOSURE(i)**

Pushes a reference to the cell contained in slot $i$ of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if $i$ is less than the length of `co_cellvars`. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

**LOAD_DEREF(i)**

Loads the cell contained in slot $i$ of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

**LOAD_CLASSDEREF(i)**

Much like **LOAD_DEREF** but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

*New in version 3.4.*

**STORE_DEREF(i)**

Stores TOS into the cell contained in slot $i$ of the cell and free variable storage.

**DELETE_DEREF(i)**

Empties the cell contained in slot $i$ of the cell and free variable storage. Used by the `del` statement.

*New in version 3.2.*

**RAISE_VARARGS(argc)**

Raises an exception. `argc` indicates the number of arguments to the `raise` statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.
CALL_FUNCTION\((argc)\)

Calls a callable object with positional arguments. \(argc\) indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. CALL_FUNCTION pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

\textit{Changed in version 3.6}: This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW\((argc)\)

Calls a callable object with positional (if any) and keyword arguments. \(argc\) indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. CALL_FUNCTION_KW pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

\textit{Changed in version 3.6}: Keyword arguments are packed in a tuple instead of a dictionary, \(argc\) indicates the total number of arguments.

CALL_FUNCTION_EX\((flags)\)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of \(flags\) is set, the top of the stack contains a mapping object containing additional keyword
arguments. Below that is an iterable object containing positional arguments and a callable object to call. BUILD_MAP_UNPACK_WITH_CALL and BUILD_TUPLE_UNPACK_WITH_CALL can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each “unpacked” and their contents passed in as keyword and positional arguments respectively. CALL_FUNCTION_EX pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

*New in version 3.6.*

**LOAD_METHOD**(namei)

Loads a method named co_names[namei] from TOS object. TOS is popped and method and TOS are pushed when interpreter can call unbound method directly. TOS will be used as the first argument (self) by CALL_METHOD. Otherwise, NULL and method is pushed (method is bound method or something else).

*New in version 3.7.*

**CALL_METHOD**(argc)

Calls a method. argc is number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with LOAD_METHOD. Positional arguments are on top of the stack. Below them, two items described in LOAD_METHOD on the stack. All of them are popped and return value is pushed.

*New in version 3.7.*

**MAKE_FUNCTION**(argc)
Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- **0x01** a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- **0x02** a dictionary of keyword-only parameters’ default values
- **0x04** an annotation dictionary
- **0x08** a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the qualified name of the function (at TOS)

**BUILD_SLICE**(*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the `slice()` built-in function for more information.

**EXTENDED_ARG**(*ext*)

Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode’s argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

**FORMAT_VALUE**(*flags*)

Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

- `(flags & 0x03) == 0x00: value` is formatted as-is.
- `(flags & 0x03) == 0x01: call `str()` on value before formatting it.`
\begin{itemize}
\item (flags & 0x03) == 0x02: call \texttt{repr()} on \textit{value} before formatting it.
\item (flags & 0x03) == 0x03: call \texttt{ascii()} on \textit{value} before formatting it.
\item (flags & 0x04) == 0x04: pop \texttt{fmt_spec} from the stack and use it, else use an empty \texttt{fmt_spec}.
\end{itemize}

Formatting is performed using \texttt{PyObject_Format()}. The result is pushed on the stack.

\textit{New in version 3.6.}

\textbf{HAVE_ARGUMENT}

This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (< \texttt{HAVE_ARGUMENT} and \geq \texttt{HAVE_ARGUMENT}, respectively).

\textit{Changed in version 3.6}: Now every instruction has an argument, but opcodes < \texttt{HAVE_ARGUMENT} ignore it. Before, only opcodes \geq \texttt{HAVE_ARGUMENT} had an argument.
 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

**dis.opname**
Sequence of operation names, indexable using the bytecode.

**dis.opmap**
Dictionary mapping operation names to bytecodes.

**dis.cmp_op**
Sequence of all compare operation names.

**dis.hasconst**
Sequence of bytecodes that access a constant.

**dis.hasfree**
Sequence of bytecodes that access a free variable (note that ‘free’ in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does not include references to global or builtin scopes).

**dis.hasname**
Sequence of bytecodes that access an attribute by name.

**dis.hasjrel**
Sequence of bytecodes that have a relative jump target.

**dis.hasjabs**
Sequence of bytecodes that have an absolute jump target.
dis. **haslocal**
Sequence of bytecodes that access a local variable.

dis. **hascompare**
Sequence of bytecodes of Boolean operations.
pickletools — Tools for pickle developers

Source code: Lib/pickletools.py

This module contains various constants relating to the intimate details of the pickle module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the pickle; ordinary users of the pickle module probably won't find the pickletools module relevant.
Command line usage

New in version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```bash
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
  0: \x80 PROTO  3
  2: K BININT1  1
  4: K BININT1  2
  6: \x86 TUPLE2
  7: q BININPUT  0
  9: . STOP
highest protocol among opcodes = 2
```

Command line options

- `-a`, `--annotate`
  Annotate each line with a short opcode description.

- `-o`, `--output=<file>`
Name of a file where the output should be written.

- **l**, **--indentlevel**=<num>
  The number of blanks by which to indent a new MARK level.

- **m**, **--memo**
  When multiple objects are disassembled, preserve memo between disassemblies.

- **p**, **--preamble**=<preamble>
  When more than one pickle file are specified, print given preamble before each disassembly.
Programmatic Interface

pickletools.dis(pickle, out=None, memo=None, indentlevel=4, annotate=0)

Outputs a symbolic disassembly of the pickle to the file-like object out, defaulting to sys.stdout. pickle can be a string or a file-like object. memo can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by MARK opcodes in the stream, are indented by indentlevel spaces. If a nonzero value is given to annotate, each opcode in the output is annotated with a short description. The value of annotate is used as a hint for the column where annotation should start.

New in version 3.2: The annotate argument.

pickletools.genops(pickle)

Provides an iterator over all of the opcodes in a pickle, returning a sequence of (opcode, arg, pos) triples. opcode is an instance of an OpcodeInfo class; arg is the decoded value, as a Python object, of the opcode's argument; pos is the position at which this opcode is located. pickle can be a string or a file-like object.

pickletools.optimize(picklestring)

Returns a new equivalent pickle string after eliminating unused PUT opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.
Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

- **formatter — Generic output formatting**
  - The Formatter Interface
  - Formatter Implementations
  - The Writer Interface
  - Writer Implementations
formatter — Generic output formatting

*Deprecated since version 3.4:* Due to lack of usage, the formatter module has been deprecated.

This module supports two interface definitions, each with multiple implementations: The `formatter` interface, and the `writer` interface which is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of “change back” operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.
The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

**formatter.AS_IS**

Value which can be used in the font specification passed to the push_font() method described below, or as the new value to any other push_property() method. Pushing the AS_IS value allows the corresponding pop_property() method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

**formatter.writer**

The writer instance with which the formatter interacts.

**formatter.end_paragraph(blanklines)**

Close any open paragraphs and insert at least blanklines before the next paragraph.

**formatter.add_line_break()**

Add a hard line break if one does not already exist. This does not break the logical paragraph.

**formatter.add_hor_rule(*args, **kw)**

Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the
writer's send_line_break() method.

**formatter.add_flow_data(data)**

Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to add_flow_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

**formatter.add_literal_data(data)**

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of data.

**formatter.add_label_data(format, counter)**

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the format value is a string, it is interpreted as a format specification for counter, which should be an integer. The result of this formatting becomes the value of the label; if format is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's send_label_data() method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `1` represents the counter value
formatter as an Arabic number, the characters 'A' and 'a' represent alphabetic representations of the counter value in upper and lower case, respectively, and 'I' and 'i' represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter. **flush_softspace()**
Send any pending whitespace buffered from a previous call to add_flowing_data() to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter. **push_alignment** (align)
Push a new alignment setting onto the alignment stack. This may be AS_IS if no change is desired. If the alignment value is changed from the previous setting, the writer's new_alignment() method is called with the align value.

formatter. **pop_alignment()**
Restore the previous alignment.

formatter. **push_font** ((size, italic, bold, teletype))
Change some or all font properties of the writer object. Properties which are not set to AS_IS are set to the values passed in while others are maintained at their current settings. The writer's new_font() method is called with the fully resolved font specification.

formatter. **pop_font()**
Restore the previous font.

formatter. **push_margin** (margin)
Increase the number of left margin indentations by one, associating the logical tag \textit{margin} with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than \texttt{AS\_IS} are not sufficient to change the margin.

\begin{verbatim}
formatter.pop_margin()
 Restore the previous margin.

formatter.push_style(*styles)
 Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including \texttt{AS\_IS} values, is passed to the writer's \texttt{new_styles()} method.

formatter.pop_style(n=1)
 Pop the last $n$ style specifications passed to \texttt{push_style()}. A tuple representing the revised stack, including \texttt{AS\_IS} values, is passed to the writer's \texttt{new_styles()} method.

formatter.set_spacing(spacing)
 Set the spacing style for the writer.

formatter.assert_line_data(flag=1)
 Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional \texttt{flag} argument can be set to false if the writer manipulations produced a hard line break at the end of the output.
\end{verbatim}
Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.NullFormatter(writer=\texttt{None})

A formatter which does nothing. If \texttt{writer} is omitted, a \texttt{NullWriter} instance is created. No methods of the writer are called by \texttt{NullFormatter} instances. Implementations should inherit from this class if implementing a writer interface but don’t need to inherit any implementation.

class formatter.AbstractFormatter(writer)

The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.
The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

**writer.flush()**

Flush any buffered output or device control events.

**writer.new_alignment(align)**

Set the alignment style. The `align` value can be any object, but by convention is a string or `None`, where `None` indicates that the writer’s “preferred” alignment should be used. Conventional `align` values are ‘left’, ‘center’, ‘right’, and ‘justify’.

**writer.new_font(font)**

Set the font style. The value of `font` will be `None`, indicating that the device’s default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The `italic`, `bold`, and `teletype` values are Boolean values specifying which of those font attributes should be used.

**writer.new_margin(margin, level)**

Set the margin level to the integer `level` and the logical tag to `margin`. Interpretation of the logical tag is at the writer’s discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of `level`. 
writer.\texttt{new\_spacing}(\textit{spacing})

Set the spacing style to \textit{spacing}.

writer.\texttt{new\_styles}(\textit{styles})

Set additional styles. The \textit{styles} value is a tuple of arbitrary values; the value \texttt{AS\_IS} should be ignored. The \textit{styles} tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.\texttt{send\_line\_break}()

Break the current line.

writer.\texttt{send\_paragraph}(\textit{blankline})

Produce a paragraph separation of at least \textit{blankline} blank lines, or the equivalent. The \textit{blankline} value will be an integer. Note that the implementation will receive a call to \texttt{send\_line\_break()} before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.\texttt{send\_hor\_rule}(*\textit{args}, **\textit{kw})

Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via \texttt{send\_line\_break()}.

writer.\texttt{send\_flowing\_data}(\textit{data})

Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.
writer.send_literal_data(data)

Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the send_formatted_data() interface.

writer.send_label_data(data)

Set data to the left of the current left margin, if possible. The value of data is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.
Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the NullWriter class.

class formatter.NullWriter
    A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.AbstractWriter
    A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.DumbWriter(file=None, maxcol=72)
    Simple writer class which writes output on the file object passed in as file or, if file is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by maxcol. This class is suitable for reflowing a sequence of paragraphs.
This chapter describes modules that are only available on MS Windows platforms.

- **msilib** — Read and write Microsoft Installer files
  - Database Objects
  - View Objects
  - Summary Information Objects
  - Record Objects
  - Errors
  - CAB Objects
  - Directory Objects
  - Features
  - GUI classes
  - Precomputed tables

- **msvcrt** — Useful routines from the MS VC++ runtime
  - File Operations
  - Console I/O
  - Other Functions

- **winreg** — Windows registry access
  - Functions
  - Constants
    - HKEY_* Constants
    - Access Rights
      - 64-bit Specific
    - Value Types
  - Registry Handle Objects

- **winsound** — Sound-playing interface for Windows
msilib — Read and write Microsoft Installer files

Source code: Lib/msilib/__init__.py

The msilib supports the creation of Microsoft Installer (.msi) files. Because these files often contain an embedded “cabinet” file (.cab), it also exposes an API to create CAB files. Support for reading .cab files is currently not implemented; read support for the .msi database is possible.

This package aims to provide complete access to all tables in an .msi file, therefore, it is a fairly low-level API. Two primary applications of this package are the distutils command bdist_msi, and the creation of Python installer package itself (although that currently uses a different version of msilib).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

msilib.FCICreate(cabname, files)

Create a new CAB file named cabname. files must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.
Callbacks to Python for the various steps of MSI creation are currently not exposed.

**msilib.**`UuidCreate()`

Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

**msilib.**`OpenDatabase(path, persist)`

Return a new database object by calling `MsiOpenDatabase`. `path` is the file name of the MSI file; `persist` can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRCREAT`, `MSIDBOPEN_READONLY`, or `MSIDBOPEN_TRANSACT`, and may include the flag `MSIDBOPEN_PATCHFILE`. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

**msilib.**`CreateRecord(count)`

Return a new record object by calling `MSICreateRecord()`. `count` is the number of fields of the record.

**msilib.**`init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)`

Create and return a new database `name`, initialize it with `schema`, and set the properties `ProductName`, `ProductCode`, `ProductVersion`, and `Manufacturer`.

`schema` must be a module object containing `tables` and `_Validation_records` attributes; typically, `msilib.schema` should be used.
The database will contain just the schema and the validation records when this function returns.

```python
text

msilib.add_data(database, table, records)
```

Add all records to the table named table in database.

The table argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be ints, strings, or instances of the Binary class.

```python
class msilib.Binary(filename)
```

Represents entries in the Binary table; inserting such an object using add_data() reads the file named filename into the table.

```python
text

msilib.add_tables(database, module)
```

Add all table content from module to database. module must contain an attribute tables listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

```python
text

msilib.add_stream(database, name, path)
```

Add the file path into the _Stream table of database, with the stream name name.

```python
text

msilib.gen_uuid()
```

Return a new UUID, in the format that MSI typically requires (i.e. in
curly braces, and with all hexdigits in upper-case).

See also: FCICreate UuidCreate UuidToString
Database Objects

Database. **OpenView**(sql)

Return a view object, by calling MSIDatabaseOpenView(). `sql` is the SQL statement to execute.

Database. **Commit**()

Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database. **GetSummaryInformation**(count)

Return a new summary information object, by calling MsiGetSummaryInformation(). `count` is the maximum number of updated values.

Database. **Close**()

Close the database object, through MsiCloseHandle().

*New in version 3.7.*

**See also:** MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle
View Objects

View. **Execute**(params)
Execute the SQL query of the view, through MSIViewExecute(). If params is not None, it is a record describing actual values of the parameter tokens in the query.

View. **GetColumnInfo**(kind)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). kind can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View. **Fetch**()
Return a result record of the query, through calling MsiViewFetch().

View. **Modify**(kind, data)
Modify the view, by calling MsiViewModify(). kind can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View. **Close**()
Close the view, through MsiViewClose().
See also: MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose
Summary Information Objects

SummaryInformation. **GetProperty**(field)

Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVISION, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation. **GetPropertyCount**()

Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation. **SetProperty**(field, value)

Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in **GetProperty()**, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation. **Persist**()

Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

See also: MsiSummaryInfoGetProperty
MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty
MsiSummaryInfoPersist
Record Objects

Record. **GetFieldCount()**
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record. **GetInteger(field)**
Return the value of *field* as an integer where possible. *field* must be an integer.

Record. **GetString(field)**
Return the value of *field* as a string where possible. *field* must be an integer.

Record. **SetString(field, value)**
Set *field* to *value* through MsiRecordSetString(). *field* must be an integer; *value* a string.

Record. **SetStream(field, value)**
Set *field* to the contents of the file named *value*, through MsiRecordSetStream(). *field* must be an integer; *value* a string.

Record. **SetInteger(field, value)**
Set *field* to *value* through MsiRecordSetInteger(). Both *field* and *value* must be an integer.

Record. **ClearData()**
Set all fields of the record to 0, through MsiRecordClearData().
See also: MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData
Errors

All wrappers around MSI functions raise MSIError; the string inside the exception will contain more detail.
CAB Objects

class msilib.CAB(name)

The class CAB represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append(full, file, logical)

Add the file with the pathname full to the CAB file, under the name logical. If there is already a file named logical, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit(database)

Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.
Directory Objects

```
class msilib.Directory(database, cab, basedir, physical, logical, default[, componentflags])
```

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through `start_component()`, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be `None`), the path to the physical directory, and a logical directory name. `default` specifies the DefaultDir slot in the directory table. `componentflags` specifies the default flags that new components get.

```
start_component(component=None, feature=None, flags=None, keyfile=None, uuid=None)
```

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no `feature` is given, the current feature is used. If no `flags` are given, the directory’s default flags are used. If no `keyfile` is given, the KeyPath is left null in the Component table.

```
add_file(file, src=None, version=None, language=None)
```

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the `src` file is specified, it is interpreted relative to the current directory. Optionally, a `version` and a `language` can be specified for the entry in the File table.
**glob(pattern, exclude=None)**

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the `exclude` list.

**remove_pyc()**

Remove `.pyc` files on uninstall.

---

**See also:** Directory Table File Table Component Table FeatureComponents Table
Features

```
class msilib.Feature(db, id, title, desc, display, level=1,
parent=None, directory=None, attributes=0)
```

Add a new record to the Feature table, using the values \( id \), \( parent.id \), \( title \), \( desc \), \( display \), \( level \), \( directory \), and \( attributes \). The resulting feature object can be passed to the start_component() method of `Directory`.

```
set_current()
```

Make this feature the current feature of `msilib`. New components are automatically added to the default feature, unless a feature is explicitly specified.

See also: Feature Table
GUI classes

`msilib` provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

```python
class msilib.Control(dlg, name)
    Base class of the dialog controls. `dlg` is the dialog object the control belongs to, and `name` is the control’s name.

event(event, argument, condition=1, ordering=None)
    Make an entry into the ControlEvent table for this control.

mapping(event, attribute)
    Make an entry into the EventMapping table for this control.

condition(action, condition)
    Make an entry into the ControlCondition table for this control.
```

```python
class msilib.RadioButtonGroup(dlg, name, property)
    Create a radio button control named `name`. `property` is the installer property that gets set when a radio button is selected.

    add(name, x, y, width, height, text, value=None)
        Add a radio button named `name` to the group, at the coordinates `x, y, width, height`, and with the label `text`. If `value` is None, it defaults to `name`.
```

```python
class msilib.Dialog(db, name, x, y, w, h, attr, title, first, default, cancel)
```
Return a new `<Dialog>` object. An entry in the `<Dialog>` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

```plaintext
control(name, type, x, y, width, height, attributes, property, text, control_next, help)
```
Return a new `<Control>` object. An entry in the `<Control>` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

```plaintext
text(name, x, y, width, height, attributes, text)
```
Add and return a `<Text>` control.

```plaintext
bitmap(name, x, y, width, height, text)
```
Add and return a `<Bitmap>` control.

```plaintext
line(name, x, y, width, height)
```
Add and return a `<Line>` control.

```plaintext
pushbutton(name, x, y, width, height, attributes, text, next_control)
```
Add and return a `<PushButton>` control.

```plaintext
radiogroup(name, x, y, width, height, attributes, property, text, next_control)
```
Add and return a `<RadioButtonGroup>` control.

```plaintext
checkbox(name, x, y, width, height, attributes, property, text, next_control)
```
Add and return a `<CheckBox>` control.
See also: Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table
Precomputed tables

**msilib** provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

**msilib.schema**

This is the standard MSI schema for MSI 2.0, with the `tables` variable providing a list of table definitions, and `_Validation_records` providing the data for MSI validation.

**msilib.sequence**

This module contains table contents for the standard sequence tables: `AdminExecuteSequence`, `AdminUISequence`, `AdvtExecuteSequence`, `InstallExecuteSequence`, and `InstallUISequence`.

**msilib.text**

This module contains definitions for the UIText and ActionText tables, for the standard installer actions.
msvcrt — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the getpass module uses this in the implementation of the getpass() function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used wherever possible.

*Changed in version 3.3:* Operations in this module now raise OSError where IOError was raised.
File Operations

`msvcrtl.locking(fd, mode, nbytes)`

Lock part of a file based on file descriptor `fd` from the C runtime. Raises `OSError` on failure. The locked region of the file extends from the current file position for `nbytes` bytes, and may continue beyond the end of the file. `mode` must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

`msvcrtl.LK_LOCK`  
`msvcrtl.LK_RLCK`

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `OSError` is raised.

`msvcrtl.LK_NBLCK`  
`msvcrtl.LK_NBRLCK`

Locks the specified bytes. If the bytes cannot be locked, `OSError` is raised.

`msvcrtl.LK_UNLCK`

Unlocks the specified bytes, which must have been previously locked.

`msvcrtl.setmode(fd, flags)`

Set the line-end translation mode for the file descriptor `fd`. To set it to text mode, `flags` should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`. 
**msvcrt.** **open_osfhandle**(*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of `os.O_APPEND`, `os.O_RDONLY`, and `os.O_TEXT`. The returned file descriptor may be used as a parameter to `os.fdopen()` to create a file object.

**msvcrt.** **get_osfhandle**(*fd*)

Return the file handle for the file descriptor *fd*. Raises `OSError` if *fd* is not recognized.
Console I/O

**msvcrtn.**\textbf{kbhit}()
Return true if a keypress is waiting to be read.

**msvcrtn.**\textbf{getch}()
Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for \texttt{Enter} to be pressed. If the pressed key was a special function key, this will return \texttt{'\000'} or \texttt{'\xe0'}; the next call will return the keycode. The \texttt{Control-C} keypress cannot be read with this function.

**msvcrtn.**\textbf{getwch}()
Wide char variant of \textbf{getch}(), returning a Unicode value.

**msvcrtn.**\textbf{getche}()
Similar to \textbf{getch}(), but the keypress will be echoed if it represents a printable character.

**msvcrtn.**\textbf{getwche}()
Wide char variant of \textbf{getche}(), returning a Unicode value.

**msvcrtn.**\textbf{putch}(\texttt{char})
Print the byte string \texttt{char} to the console without buffering.

**msvcrtn.**\textbf{putwch}(\texttt{unicode_char})
Wide char variant of \textbf{putch}(), accepting a Unicode value.

**msvcrtn.**\textbf{ungetch}(\texttt{char})
Cause the byte string \texttt{char} to be “pushed back” into the console
buffer; it will be the next character read by `getch()` or `getche()`.

`msvcrt.ungetwch(unicode_char)`

Wide char variant of `ungetch()`, accepting a Unicode value.
Other Functions

`msvcrt.heapmin()`

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises `OSError`.
These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

*Changed in version 3.3:* Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.
Functions

This module offers the following functions:

\texttt{winreg.\texttt{CloseKey}(hkey)} 
Closes a previously opened registry key. The \texttt{hkey} argument specifies a previously opened key.

\textbf{Note:} If \texttt{hkey} is not closed using this method (or via \texttt{hkey.Close()}), it is closed when the \texttt{hkey} object is destroyed by Python.

\texttt{winreg.\texttt{ConnectRegistry}(computer\_name, key)} 
Establishes a connection to a predefined registry handle on another computer, and returns a handle object.

\textit{computer\_name} is the name of the remote computer, of the form \texttt{r"\\computername"}. If \texttt{None}, the local computer is used.

\textit{key} is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an \texttt{OSError} exception is raised.

\textit{Changed in version 3.3}: See \texttt{above}.

\texttt{winreg.\texttt{CreateKey}(key, sub\_key)} 
Creates or opens the specified key, returning a handle object.

\textit{key} is an already open key, or one of the predefined \texttt{HKEY\_* constants}.

\textit{sub\_key} is a string that names the key this method opens or
creates.

If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

*Changed in version 3.3:* See above.

```python
winreg.CreateKeyEx(key, sub_key, reserved=0, access=KEY_WRITE)
```

Creates or opens the specified key, returning a handle object.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that names the key this method opens or creates.

`reserved` is a reserved integer, and must be zero. The default is zero.

`access` is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See `Access Rights` for other allowed values.

If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.
If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an OSError exception is raised.

*New in version 3.2.*

*Changed in version 3.3:* See [above](#).

```python
winreg.DeleteKey(key, sub_key)
```

Deletes the specified key.

- `key` is an already open key, or one of the predefined HKEY_* constants.
- `sub_key` is a string that must be a subkey of the key identified by the `key` parameter. This value must not be None, and the key may not have subkeys.

*This method can not delete keys with subkeys.*

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an OSError exception is raised.

*Changed in version 3.3:* See [above](#).

```python
winreg.DeleteKeyEx(key, sub_key, access=KEY_WOW64_64KEY, reserved=0)
```

Deletes the specified key.

**Note:** The `DeleteKeyEx()` function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.
Key is an already open key, or one of the predefined HKEY_* constants.

Sub_key is a string that must be a subkey of the key identified by the key parameter. This value must not be None, and the key may not have subkeys.

Reserved is a reserved integer, and must be zero. The default is zero.

Access is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_WOW64_64KEY. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an OSError exception is raised.

On unsupported Windows versions, NotImplementedError is raised.

New in version 3.2.

Changed in version 3.3: See above.

Winreg. DeleteValue(key, value)

Removes a named value from a registry key.

Key is an already open key, or one of the predefined HKEY_* constants.

Value is a string that identifies the value to remove.

Winreg. EnumKey(key, index)
Enumerates subkeys of an open registry key, returning a string.

*key* is an already open key, or one of the predefined HKEY_* constants.

*index* is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an OSError exception is raised, indicating, no more values are available.

*Changed in version 3.3:* See above.

```
winreg. EnumValue(key, index)
```

Enumerates values of an open registry key, returning a tuple.

*key* is an already open key, or one of the predefined HKEY_* constants.

*index* is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an OSError exception is raised, indicating no more values.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A string that identifies the value name</td>
</tr>
<tr>
<td>1</td>
<td>An object that holds the value data, and whose type depends on the underlying registry type</td>
</tr>
<tr>
<td>2</td>
<td>An integer that identifies the type of the value data (see table in docs for SetValueEx())</td>
</tr>
</tbody>
</table>
Changed in version 3.3: See above.

**winreg. ExpandEnvironmentStrings**(str)
Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ`:

```python
>>> ExpandEnvironmentStrings('%windir%')
'C:\Windows'
```

**winreg. FlushKey**(key)
Writes all the attributes of a key to the registry.

*key* is an already open key, or one of the predefined `HKEY_*` constants.

It is not necessary to call `FlushKey()` to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike `CloseKey()`, the `FlushKey()` method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

**Note:** If you don’t know whether a `FlushKey()` call is required, it probably isn’t.

**winreg. LoadKey**(key, sub_key, file_name)
Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

*key* is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`. 
**sub_key** is a string that identifies the subkey to load.

**file_name** is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If **key** is a handle returned by `ConnectRegistry()`, then the path specified in **file_name** is relative to the remote computer.

```c
winreg.OpenKey(key, sub_key, reserved=0, access=KEY_READ)
winreg.OpenKeyEx(key, sub_key, reserved=0, access=KEY_READ)
```

Opens the specified key, returning a handle object.

**key** is an already open key, or one of the predefined HKEY_* constants.

**sub_key** is a string that identifies the sub_key to open.

**reserved** is a reserved integer, and must be zero. The default is zero.

**access** is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ. See Access Rights for other allowed values.

The result is a new handle to the specified key.
If the function fails, `OSError` is raised.

*Changed in version 3.2:* Allow the use of named arguments.

*Changed in version 3.3:* See above.

**winreg.** `QueryInfoKey(key)`

Returns information about a key, as a tuple.

*key* is an already open key, or one of the predefined `HKEY_*` constants.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An integer giving the number of sub keys this key has.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the number of values this key has.</td>
</tr>
<tr>
<td>2</td>
<td>An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.</td>
</tr>
</tbody>
</table>

**winreg.** `QueryValue(key, sub_key)`

Retrieves the unnamed value for a key, as a string.

*key* is an already open key, or one of the predefined `HKEY_*` constants.

*sub_key* is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the `SetValue()` method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL
name. But the underlying API call doesn’t return the type, so always use `QueryValueEx()` if possible.

`winreg.QueryValueEx(key, value_name)`

Retrieves the type and data for a specified value name associated with an open registry key.

`key` is an already open key, or one of the predefined `HKEY_* constants`.

`value_name` is a string indicating the value to query.

The result is a tuple of 2 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value of the registry item.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the registry type for this value (see table in docs for <code>SetValueEx()</code>)</td>
</tr>
</tbody>
</table>

`winreg.SaveKey(key, file_name)`

Saves the specified key, and all its subkeys to the specified file.

`key` is an already open key, or one of the predefined `HKEY_* constants`.

`file_name` is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If `key` represents a key on a remote computer, the path described by `file_name` is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security
privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for security_attributes to the API.

```
winreg.SetValue(key, sub_key, type, value)

Associates a value with a specified key.
```

- **key** is an already open key, or one of the predefined HKEY_* constants.
- **sub_key** is a string that names the subkey with which the value is associated.
- **type** is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the SetValueEx() function for support for other data types.
- **value** is a string that specifies the new value.

If the key specified by the sub_key parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

```
winreg.SetValueEx(key, value_name, reserved, type, value)

Stores data in the value field of an open registry key.
```
key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See Value Types for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with KEY_SET_VALUE access.

To open the key, use the CreateKey() or OpenKey() methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

winreg.DisableReflectionKey(key)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise NotImplemented if executed on a 32-bit operating system.
If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

\texttt{winreg.\texttt{EnableReflectionKey(key)}}

Restores registry reflection for the specified disabled key.

\textit{key} is an already open key, or one of the predefined \texttt{HKEY_* constants}.

Will generally raise \texttt{NotImplemented} if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

\texttt{winreg.\texttt{QueryReflectionKey(key)}}

Determines the reflection state for the specified key.

\textit{key} is an already open key, or one of the predefined \texttt{HKEY_* constants}.

Returns \texttt{True} if reflection is disabled.

Will generally raise \texttt{NotImplemented} if executed on a 32-bit operating system.
Constants

The following constants are defined for use in many _winreg functions.

HKEY_* Constants

winreg.HKEY_CLASSES_ROOT
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

winreg.HKEY_CURRENT_USER
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

winreg.HKEY_LOCAL_MACHINE
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

winreg.HKEY_USERS
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

winreg.HKEY_PERFORMANCE_DATA
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry;
the registry functions cause the system to collect the data from its source.

\texttt{winreg.HKEY_CURRENT_CONFIG}
Contains information about the current hardware profile of the local computer system.

\texttt{winreg.HKEY_DYN_DATA}
This key is not used in versions of Windows after 98.

\textbf{Access Rights}

For more information, see Registry Key Security and Access.

\texttt{winreg.KEY_ALL_ACCESS}
Combines the \texttt{STANDARD_RIGHTS_REQUIRED}, \texttt{KEY_QUERY_VALUE}, \texttt{KEY_SET_VALUE}, \texttt{KEY_CREATE_SUB_KEY}, \texttt{KEY_ENUMERATE_SUB_KEYS}, \texttt{KEY_NOTIFY}, and \texttt{KEY_CREATE_LINK} access rights.

\texttt{winreg.KEY_WRITE}
Combines the \texttt{STANDARD_RIGHTS_WRITE}, \texttt{KEY_SET_VALUE}, and \texttt{KEY_CREATE_SUB_KEY} access rights.

\texttt{winreg.KEY_READ}
Combines the \texttt{STANDARD_RIGHTS_READ}, \texttt{KEY_QUERY_VALUE}, \texttt{KEY_ENUMERATE_SUB_KEYS}, and \texttt{KEY_NOTIFY} values.

\texttt{winreg.KEY_EXECUTE}
Equivalent to \texttt{KEY_READ}.

\texttt{winreg.KEY_QUERY_VALUE}
Required to query the values of a registry key.

`winreg.KEY_SET_VALUE`
Required to create, delete, or set a registry value.

`winreg.KEY_CREATE_SUB_KEY`
Required to create a subkey of a registry key.

`winreg.KEY_ENUMERATE_SUB_KEYS`
Required to enumerate the subkeys of a registry key.

`winreg.KEY_NOTIFY`
Required to request change notifications for a registry key or for subkeys of a registry key.

`winreg.KEY_CREATE_LINK`
Reserved for system use.

64-bit Specific

For more information, see [Accessing an Alternate Registry View](#).

`winreg.KEY_WOW64_64KEY`
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

`winreg.KEY_WOW64_32KEY`
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see [Registry Value Types](#).
winreg.**REG_BINARY**

Binary data in any form.

winreg.**REG_DWORD**

32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**

A 32-bit number in little-endian format. Equivalent to **REG_DWORD**.

winreg.**REG_DWORD_BIG_ENDIAN**

A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**

Null-terminated string containing references to environment variables (\%PATH\%).

winreg.**REG_LINK**

A Unicode symbolic link.

winreg.**REG_MULTI_SZ**

A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**

No defined value type.

winreg.**REG_QWORD**

A 64-bit number.

_New in version 3.6._

winreg.**REG_QWORD_LITTLE_ENDIAN**

A 64-bit number in little-endian format. Equivalent to **REG_QWORD**.
New in version 3.6.

winreg. **REGRESOURCE_LIST**
A device-driver resource list.

winreg. **REGFULLRESOURCE_DESCRIPTOR**
A hardware setting.

winreg. **REGRESOURCE_REQUIREMENTS_LIST**
A hardware resource list.

winreg. **REG_SZ**
A null-terminated string.
Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the `Close()` method on the object, or the `CloseKey()` function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```python
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in `int()` function), in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

```python
PyHKEY.Close()
```

Closes the underlying Windows handle.
If the handle is already closed, no error is raised.

**PyHKEY.** `Detach()`  
Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

**PyHKEY.** `__enter__()`  
**PyHKEY.** `__exit__(exc_info)`

The `HKEY` object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement:

```python
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...
    # work with key
```

will automatically close `key` when control leaves the `with` block.
winsound — Sound-playing interface for Windows

The **winsound** module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

**winsound.** **Beep**(frequency, duration)

Beep the PC’s speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, **RuntimeError** is raised.

**winsound.** **PlaySound**(sound, flags)

Call the underlying PlaySound() function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a **bytes-like object**, or **None**. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is **None**, any currently playing waveform sound is stopped. If the system indicates an error, **RuntimeError** is raised.

**winsound.** **MessageBeep**(type=MB_OK)

Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are -1, **MB_ICONASTERISK**, **MB_ICONEXCLAMATION**, **MB_ICONHAND**, **MB_ICONHAND**,
MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a “simple beep”; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, RuntimeError is raised.

winsound. SND_FILENAME

The sound parameter is the name of a WAV file. Do not use with SND_ALIAS.

winsound. SND_ALIAS

The sound parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless SND_NODEFAULT is also specified. If no default sound is registered, raise RuntimeError. Do not use with SND_FILENAME.

All Win32 systems support at least the following; most systems support many more:

<table>
<thead>
<tr>
<th>PlaySound() name</th>
<th>Corresponding Control Panel Sound name</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SystemAsterisk'</td>
<td>Asterisk</td>
</tr>
<tr>
<td>'SystemExclamation'</td>
<td>Exclamation</td>
</tr>
<tr>
<td>'SystemExit'</td>
<td>Exit Windows</td>
</tr>
<tr>
<td>'SystemHand'</td>
<td>Critical Stop</td>
</tr>
<tr>
<td>'SystemQuestion'</td>
<td>Question</td>
</tr>
</tbody>
</table>

For example:

```python
import winsound
# Play Windows exit sound.
```
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is
# "*" probably isn't the registered name of any s
winsound.PlaySound("*", winsound.SND_ALIAS)

winsound.SND_LOOP
Play the sound repeatedly. The SND_ASYNC flag must also be
used to avoid blocking. Cannot be used with SND_MEMORY.

winsound.SND_MEMORY
The sound parameter to PlaySound() is a memory image of a
WAV file, as a bytes-like object.

Note: This module does not support playing from a memory
image asynchronously, so a combination of this flag and
SND_ASYNC will raise RuntimeError.

winsound.SND_PURGE
Stop playing all instances of the specified sound.

Note: This flag is not supported on modern Windows platforms.

winsound.SND_ASYNC
Return immediately, allowing sounds to play asynchronously.

winsound.SND_NODEFAULT
If the specified sound cannot be found, do not play the system
default sound.

winsound.SND_NOSTOP
Do not interrupt sounds currently playing.
winsound. **SND_NOWAIT**

Return immediately if the sound driver is busy.

---

**Note:** This flag is not supported on modern Windows platforms.

winsound. **MB_ICONASTERISK**

Play the SystemDefault sound.

winsound. **MB_ICONEXCLAMATION**

Play the SystemExclamation sound.

winsound. **MB_ICONHAND**

Play the SystemHand sound.

winsound. **MB_ICONQUESTION**

Play the SystemQuestion sound.

winsound. **MB_OK**

Play the SystemDefault sound.
Unix Specific Services

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here's an overview:

- **posix** — The most common POSIX system calls
  - Large File Support
  - Notable Module Contents
- **pwd** — The password database
- **spwd** — The shadow password database
- **grp** — The group database
- **crypt** — Function to check Unix passwords
  - Hashing Methods
  - Module Attributes
  - Module Functions
  - Examples
- **termios** — POSIX style tty control
  - Example
- **tty** — Terminal control functions
- **pty** — Pseudo-terminal utilities
  - Example
- **fcntl** — The fcntl and ioctl system calls
- **pipes** — Interface to shell pipelines
  - Template Objects
- **resource** — Resource usage information
  - Resource Limits
  - Resource Usage
- **nis** — Interface to Sun’s NIS (Yellow Pages)
- **syslog** — Unix syslog library routines
  - Examples
    - Simple example
**posix** — The most common POSIX system calls

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

Do not import this module directly. Instead, import the module `os`, which provides a portable version of this interface. On Unix, the `os` module provides a superset of the `posix` interface. On non-Unix operating systems the `posix` module is not available, but a subset is always available through the `os` interface. Once `os` is imported, there is no performance penalty in using it instead of `posix`. In addition, `os` provides some additional functionality, such as automatically calling `putenv()` when an entry in `os.environ` is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise `OSError`.
Large File Support

Several operating systems (including AIX, HP-UX, Irix and Solaris) provide support for files that are larger than 2 GiB from a C programming model where int and long are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as large files.

Large file support is enabled in Python when the size of an off_t is larger than a long and the long long type is available and is at least as large as an off_t. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS"
./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS=' -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64'
./configure
```
Notable Module Contents

In addition to many functions described in the `os` module documentation, `posix` defines the following data item:

`posix.environ`  
A dictionary representing the string environment at the time the interpreter was started. Keys and values are bytes on Unix and strings on Windows. For example, `environ[b'HOME']` (or `environ['HOME']` on Windows) is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`; if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`.

*Changed in version 3.2:* On Unix, keys and values are bytes.

**Note:** The `os` module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating `os.environ` will render this dictionary obsolete. Use of the `os` module version of this is recommended over direct access to the `posix` module.
pwd — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the passwd structure (Attribute field below, see <pwd.h>):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pw_name</td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td>pw_passwd</td>
<td>Optional encrypted password</td>
</tr>
<tr>
<td>2</td>
<td>pw_uid</td>
<td>Numerical user ID</td>
</tr>
<tr>
<td>3</td>
<td>pw_gid</td>
<td>Numerical group ID</td>
</tr>
<tr>
<td>4</td>
<td>pw_gecos</td>
<td>User name or comment field</td>
</tr>
<tr>
<td>5</td>
<td>pw_dir</td>
<td>User home directory</td>
</tr>
<tr>
<td>6</td>
<td>pw_shell</td>
<td>User command interpreter</td>
</tr>
</tbody>
</table>

The uid and gid items are integers, all others are strings. KeyError is raised if the entry asked for cannot be found.

Note: In traditional Unix the field pw_passwd usually contains a password encrypted with a DES derived algorithm (see module crypt). However most modern unices use a so-called shadow password system. On those unices the pw_passwd field only contains an asterisk ('*') or the letter 'x' where the encrypted password is stored in a file /etc/shadow which is not world readable. Whether the pw_passwd field contains anything useful is
system-dependent. If available, the `spwd` module should be used where access to the encrypted password is required.

It defines the following items:

```python
pwd.getpwuid(uid)
```

Return the password database entry for the given numeric user ID.

```python
pwd.getpwnam(name)
```

Return the password database entry for the given user name.

```python
pwd.getpwall()
```

Return a list of all available password database entries, in arbitrary order.

**See also:**

**Module** `grp`  
An interface to the group database, similar to this.

**Module** `pwd`  
An interface to the shadow password database, similar to this.
**spwd** — The shadow password database

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `spwd` structure (Attribute field below, see `<shadow.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>sp_namp</code></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><code>sp_pwdp</code></td>
<td>Encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><code>sp_lstchg</code></td>
<td>Date of last change</td>
</tr>
<tr>
<td>3</td>
<td><code>sp_min</code></td>
<td>Minimal number of days between changes</td>
</tr>
<tr>
<td>4</td>
<td><code>sp_max</code></td>
<td>Maximum number of days between changes</td>
</tr>
<tr>
<td>5</td>
<td><code>sp_warn</code></td>
<td>Number of days before password expires to warn user about it</td>
</tr>
<tr>
<td>6</td>
<td><code>sp_inact</code></td>
<td>Number of days after password expires until account is disabled</td>
</tr>
<tr>
<td>7</td>
<td><code>sp_expire</code></td>
<td>Number of days since 1970-01-01 when account expires</td>
</tr>
<tr>
<td>8</td>
<td><code>sp_flag</code></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
The sp_namp and sp_pwdp items are strings, all others are integers. KeyError is raised if the entry asked for cannot be found.

The following functions are defined:

spwd.getspnam(name)

Return the shadow password database entry for the given user name.

Changed in version 3.6: Raises a PermissionError instead of KeyError if the user doesn’t have privileges.

spwd.getspall()

Return a list of all available shadow password database entries, in arbitrary order.

See also:

Module grp

An interface to the group database, similar to this.

Module pwd

An interface to the normal password database, similar to this.
The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the group structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gr_name</td>
<td>the name of the group</td>
</tr>
<tr>
<td>1</td>
<td>gr_passwd</td>
<td>the (encrypted) group password; often empty</td>
</tr>
<tr>
<td>2</td>
<td>gr_gid</td>
<td>the numerical group ID</td>
</tr>
<tr>
<td>3</td>
<td>gr_mem</td>
<td>all the group member's user names</td>
</tr>
</tbody>
</table>

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a gr_name that starts with a + or - is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid()`.)

It defines the following items:

**grp.getgrgid(gid)**

Return the group database entry for the given numeric group ID. **KeyError** is raised if the entry asked for cannot be found.
Deprecation since version 3.6: Since Python 3.6 the support of non-integer arguments like floats or strings in `getgrgid()` is deprecated.

**grp.** `getgrnam(name)`

Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.

**grp.** `getgrall()`

Return a list of all available group entries, in arbitrary order.

See also:

- **Module** `pwd`
  
  An interface to the user database, similar to this.

- **Module** `spwd`
  
  An interface to the shadow password database, similar to this.
crypt — Function to check Unix passwords

Source code: Lib/crypt.py

This module implements an interface to the `crypt(3)` routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the `crypt(3)` routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.
Hashing Methods

*New in version 3.3.*

The `crypt` module defines the list of hashing methods (not all methods are available on all platforms):

- **crypt.** `METHOD_SHA512`  
  A Modular Crypt Format method with 16 character salt and 86 character hash based on the SHA-512 hash function. This is the strongest method.

- **crypt.** `METHOD_SHA256`  
  Another Modular Crypt Format method with 16 character salt and 43 character hash based on the SHA-256 hash function.

- **crypt.** `METHOD_BLOWFISH`  
  Another Modular Crypt Format method with 22 character salt and 31 character hash based on the Blowfish cipher.  

  *New in version 3.7.*

- **crypt.** `METHOD_MD5`  
  Another Modular Crypt Format method with 8 character salt and 22 character hash based on the MD5 hash function.

- **crypt.** `METHOD_CRYPT`  
  The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.
Module Attributes

New in version 3.3.

crypt.methods
A list of available password hashing algorithms, as crypt.METHOD_* objects. This list is sorted from strongest to weakest.
Module Functions

The crypt module defines the following functions:

**crypt.crypt**(word, salt=None)

`word` will usually be a user's password as typed at a prompt or in a graphical interface. The optional `salt` is either a string as returned from `mksalt()`, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If `salt` is not provided, the strongest method will be used (as returned by `methods()`).

Checking a password is usually done by passing the plain-text password as `word` and the full results of a previous `crypt()` call, which should be the same as the results of this call.

`salt` (either a random 2 or 16 character string, possibly prefixed with `$digit$` to indicate the method) which will be used to perturb the encryption algorithm. The characters in `salt` must be in the set `.[./a-zA-Z0-9]`, with the exception of Modular Crypt Format which prefixes a `$digit$`.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few crypt(3) extensions allow different values, with different sizes in the `salt`, it is recommended to use the full crypted password as salt when checking for a password.

*Changed in version 3.3:* Accept `crypt.METHOD_*` values in addition to strings for `salt`. 
crypt.mksalt(method=None, *, rounds=None)

Return a randomly generated salt of the specified method. If no method is given, the strongest method available as returned by methods() is used.

The return value is a string suitable for passing as the salt argument to crypt().

rounds specifies the number of rounds for METHOD_SHA256, METHOD_SHA512 and METHOD_BLOWFISH. For METHOD_SHA256 and METHOD_SHA512 it must be an integer between 1000 and 999_999_999, the default is 5000. For METHOD_BLOWFISH it must be a power of two between 16 ($2^4$) and 2_147_483_648 ($2^{31}$), the default is 4096 ($2^{12}$).

New in version 3.3.

Changed in version 3.7: Added the rounds parameter.
Examples

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. \texttt{hmac.compare_digest()} is suitable for this purpose):

```python
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd ==
            raise ValueError('no support for shadow
    cleartext = getpass.getpass()
    return compare_hash(crypt.crypt(cleartext, c
else:
    return True
```

To generate a hash of a password using the strongest available method and check it against the original:

```python
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, h
raise ValueError("hashed version doesn't validate
```
**termios — POSIX style tty control**

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see `termios(3)` Unix manual page. It is only available for those Unix versions that support POSIX `termios` style tty I/O control configured during installation.

All functions in this module take a file descriptor `fd` as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

**termios.tcgetattr(fd)**

Return a list containing the tty attributes for file descriptor `fd`, as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where `cc` is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the `cc` array must be done using the symbolic constants defined in the `termios` module.

**termios.tcsetattr(fd, when, attributes)**
Set the tty attributes for file descriptor `fd` from the `attributes`, which is a list like the one returned by `tcgetattr()`. The `when` argument determines when the attributes are changed: TCSANOW to change immediately, TCSADRAIN to change after transmitting all queued output, or TCSAFLUSH to change after transmitting all queued output and discarding all queued input.

**termios.tcsendbreak(fd, duration)**
Send a break on file descriptor `fd`. A zero `duration` sends a break for 0.25–0.5 seconds; a nonzero `duration` has a system dependent meaning.

**termios.tcdrain(fd)**
Wait until all output written to file descriptor `fd` has been transmitted.

**termios.tcflush(fd, queue)**
Discard queued data on file descriptor `fd`. The `queue` selector specifies which queue: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

**termios.tcflow(fd, action)**
Suspend or resume input or output on file descriptor `fd`. The `action` argument can be TCOFF to suspend output, TCOON to restart output, TCIOFF to suspend input, or TCION to restart input.

**See also:**

**Module tty**
Convenience functions for common terminal control operations.
Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate `tcgetattr()` call and a `try ... finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt="Password: "):  
    import termios, sys  
    fd = sys.stdin.fileno()  
    old = termios.tcgetattr(fd)  
    new = termios.tcgetattr(fd)  
    try:  
        termios.tcsetattr(fd, termios.TCSADRAIN, new)  
        passwd = input(prompt)  
    finally:  
        termios.tcsetattr(fd, termios.TCSADRAIN, old)  
    return passwd
```
The `tty` module defines functions for putting the tty into cbreak and raw modes.

Because it requires the `termios` module, it will work only on Unix.

The `tty` module defines the following functions:

- **`tty.setraw(fd, when=termios.TCSAFLUSH)`**
  Change the mode of the file descriptor `fd` to raw. If `when` is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`.

- **`tty.setcbreak(fd, when=termios.TCSAFLUSH)`**
  Change the mode of file descriptor `fd` to cbreak. If `when` is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`.

### See also:

- **Module `termios`**
  Low-level terminal control interface.
pty — Pseudo-terminal utilities

Source code: Lib/pty.py

The pty module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Because pseudo-terminal handling is highly platform dependent, there is code to do it only for Linux. (The Linux code is supposed to work on other platforms, but hasn’t been tested yet.)

The pty module defines the following functions:

pty.fork()

Fork. Connect the child’s controlling terminal to a pseudo-terminal. Return value is \((\text{pid, fd})\). Note that the child gets \text{pid} 0, and the \text{fd} is invalid. The parent’s return value is the \text{pid} of the child, and \text{fd} is a file descriptor connected to the child’s controlling terminal (and also to the child’s standard input and output).

pty.openpty()

Open a new pseudo-terminal pair, using \text{os.openpty()} if possible, or emulation code for generic Unix systems. Return a pair of file descriptors \((\text{master, slave})\), for the master and the slave end, respectively.

pty.spawn(\text{argv}, [\text{master_read}, [\text{stdin_read}]])

Spawn a process, and connect its controlling terminal with the current process’s standard io. This is often used to baffle programs
which insist on reading from the controlling terminal.

The functions `master_read` and `stdin_read` should be functions which read from a file descriptor. The defaults try to read 1024 bytes each time they are called.

*Changed in version 3.4:* `spawn()` now returns the status value from `os.waitpid()` on the child process.
Example

The following program acts like the Unix command `script(1)`, using a pseudo-terminal to record all input and output of a terminal session in a “typescript”.

```python
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true',
parser.add_argument('-p', dest='use_python', action=,
parser.add_argument('filename', nargs='?', default='typescript',
options = parser.parse_args()

shell = sys.executable if options.use_python else os
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime)

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.ctime)
print('Script done, file is', filename)
```
© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
**fcntl** — The fcntl and ioctl system calls

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and ioctl() Unix routines. For a complete description of these calls, see fcntl(2) and ioctl(2) Unix manual pages.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an `io.IOBase` object, such as `sys.stdin` itself, which provides a `fileno()` that returns a genuine file descriptor.

*Changed in version 3.3:* Operations in this module used to raise an IOError where they now raise an OSError.

The module defines the following functions:

```python
fcntl.fcntl(fd, cmd, arg=0)
```

Perform the operation *cmd* on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the `fcntl` module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a `bytes` object. With an integer value, the return value of this function is the integer return value of the C fcntl() call. When the argument is bytes it represents a binary structure, e.g. created by `struct.pack()`. The binary data is copied to a
buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a `bytes` object. The length of the returned object will be the same as the length of the `arg` argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the `fcntl()` fails, an `OSError` is raised.

```
fcntl.ioctl(fd, request, arg=0, mutate_flag=True)
```

This function is identical to the `fcntl()` function, except that the argument handling is even more complicated.

The `request` parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the `request` argument can be found in the `termios` module, under the same names as used in the relevant C header files.

The `arg` parameter can be one of an integer, an object supporting the read-only buffer interface (like `bytes`) or an object supporting the read-write buffer interface (like `bytearray`).

In all but the last case, behaviour is as for the `fcntl()` function.

If a mutable buffer is passed, then the behaviour is determined by the value of the `mutate_flag` parameter.

If it is false, the buffer’s mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.
If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying *ioctl()* system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the *ioctl()*.

This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If the *ioctl()* fails, an **OSError** exception is raised.

An example:

```python
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP))
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, True)
0
>>> buf
array('h', [13341])
```

**fcntl.flock(fd, operation)**

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using *fcntl()*.)

If the *flock()* fails, an **OSError** exception is raised.

**fcntl.lockf(fd, cmd, len=0, start=0, whence=0)**

This is essentially a wrapper around the *fcntl()* locking calls. *fd*
is the file descriptor of the file to lock or unlock, and \textit{cmd} is one of the following values:

- \texttt{LOCK\_UN} – unlock
- \texttt{LOCK\_SH} – acquire a shared lock
- \texttt{LOCK\_EX} – acquire an exclusive lock

When \textit{cmd} is \texttt{LOCK\_SH} or \texttt{LOCK\_EX}, it can also be bitwise ORed with \texttt{LOCK\_NB} to avoid blocking on lock acquisition. If \texttt{LOCK\_NB} is used and the lock cannot be acquired, an \texttt{OSError} will be raised and the exception will have an \texttt{errno} attribute set to \texttt{EACCES} or \texttt{EAGAIN} (depending on the operating system; for portability, check for both values). On at least some systems, \texttt{LOCK\_EX} can only be used if the file descriptor refers to a file opened for writing.

\textit{len} is the number of bytes to lock, \textit{start} is the byte offset at which the lock starts, relative to \textit{whence}, and \textit{whence} is as with \texttt{io.IOBase.seek()}, specifically:

- \texttt{0} – relative to the start of the file (\texttt{os.SEEK\_SET})
- \texttt{1} – relative to the current buffer position (\texttt{os.SEEK\_CUR})
- \texttt{2} – relative to the end of the file (\texttt{os.SEEK\_END})

The default for \textit{start} is \texttt{0}, which means to start at the beginning of the file. The default for \textit{len} is \texttt{0} which means to lock to the end of the file. The default for \textit{whence} is also \texttt{0}.

Examples (all on a SVR4 compliant system):

```python
import struct, fcntl, os
f = open(...)  
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)  
lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0)
```
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)

Note that in the first example the return value variable `rv` will hold an integer value; in the second example it will hold a `bytes` object. The structure lay-out for the `lockdata` variable is system dependent — therefore using the `flock()` call may be better.

See also:

Module `os`

If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.
The `pipes` module defines a class to abstract the concept of a `pipeline` — a sequence of converters from one file to another.

Because the module uses `/bin/sh` command lines, a POSIX or compatible shell for `os.system()` and `os.popen()` is required.

The `pipes` module defines the following class:

```python
class pipes.Template
    An abstraction of a pipeline.
```

Example:

```python
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```
Template Objects

Template objects following methods:

Template. **reset()**

    Restore a pipeline template to its initial state.

Template. **clone()**

    Return a new, equivalent, pipeline template.

Template. **debug(flag)**

    If `flag` is true, turn debugging on. Otherwise, turn debugging off.
    When debugging is on, commands to be executed are printed, and
    the shell is given `set -x` command to be more verbose.

Template. **append(cmd, kind)**

    Append a new action at the end. The `cmd` variable must be a valid
    bourne shell command. The `kind` variable consists of two letters.

    The first letter can be either of `-` (which means the command
    reads its standard input), `'f'` (which means the commands reads
    a given file on the command line) or `'. '` (which means the
    commands reads no input, and hence must be first.)

    Similarly, the second letter can be either of `-` (which means the
    command writes to standard output), `'f'` (which means the
    command writes a file on the command line) or `'. '` (which means
    the command does not write anything, and hence must be last.)

Template. **prepend(cmd, kind)**

    Add a new action at the beginning. See **append()** for
    explanations of the arguments.
Template. **open**(file, mode)

Return a file-like object, open to file, but read from or written to by the pipeline. Note that only one of 'r', 'w' may be given.

Template. **copy**(infile, outfile)

Copy infile to outfile through the pipe.
resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An OSError is raised on syscall failure.

exception resource.error
  A deprecated alias of OSError.

  Changed in version 3.3: Following PEP 3151, this class was made an alias of OSError.
Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

**resource.**RLIM_INFINITY

Constant used to represent the limit for an unlimited resource.

**resource.**getrlimit(resource)

Returns a tuple `(soft, hard)` with the current soft and hard limits of `resource`. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

**resource.**setrlimit(resource, limits)

Sets new limits of consumption of `resource`. The `limits` argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of `RLIM_INFINITY` can be used to request a limit that is unlimited.

Raises `ValueError` if an invalid resource is specified, if the new
soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of RLIM_INFINITY when the hard or system limit for that resource is not unlimited will result in a ValueError. A process with the effective UID of super-user can request any valid limit value, including unlimited, but ValueError will still be raised if the requested limit exceeds the system imposed limit.

setrlimit may also raise error if the underlying system call fails.

resource. prlimit(pid, resource[, limits])

Combines setrlimit() and getrlimit() in one function and supports to get and set the resources limits of an arbitrary process. If pid is 0, then the call applies to the current process. resource and limits have the same meaning as in setrlimit(), except that limits is optional.

When limits is not given the function returns the resource limit of the process pid. When limits is given the resource limit of the process is set and the former resource limit is returned.

Raises ProcessLookupError when pid can’t be found and PermissionError when the user doesn’t have CAP_SYS_RESOURCE for the process.

Availability: Linux 2.6.36 or later with glibc 2.13 or later.

New in version 3.4.

These symbols define resources whose consumption can be controlled using the setrlimit() and getrlimit() functions described below. The values of these symbols are exactly the
constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

**resource.** **RLIMIT_CORE**

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

**resource.** **RLIMIT_CPU**

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a SIGXCPU signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

**resource.** **RLIMITFSIZE**

The maximum size of a file which the process may create.

**resource.** **RLIMIT_DATA**

The maximum size (in bytes) of the process’s heap.

**resource.** **RLIMIT_STACK**

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

**resource.** **RLIMIT_RSS**

The maximum resident set size that should be made available to
the process.

resource. **RLIMIT_NPROC**
The maximum number of processes the current process may create.

resource. **RLIMIT_NOFILE**
The maximum number of open file descriptors for the current process.

resource. **RLIMIT_OFILE**
The BSD name for **RLIMIT_NOFILE**.

resource. **RLIMIT_MEMLOCK**
The maximum address space which may be locked in memory.

resource. **RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

resource. **RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.

resource. **RLIMIT_MSGQUEUE**
The number of bytes that can be allocated for POSIX message queues.

**Availability**: Linux 2.6.8 or later.

**New in version 3.4.**

resource. **RLIMIT_NICE**
The ceiling for the process's nice level (calculated as 20 - rlim_cur).
Availability: Linux 2.6.12 or later.

New in version 3.4.

Resource. **RLIMIT_RTPRIO**
The ceiling of the real-time priority.

Availability: Linux 2.6.12 or later.

New in version 3.4.

Resource. **RLIMIT_RTTIME**
The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability: Linux 2.6.25 or later.

New in version 3.4.

Resource. **RLIMIT_SIGPENDING**
The number of signals which the process may queue.

Availability: Linux 2.6.8 or later.

New in version 3.4.

Resource. **RLIMIT_SBSIZE**
The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability: FreeBSD 9 or later.

New in version 3.4.
resource. **RLIMIT_SWAP**

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id’s processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see *tuning(7)* for a complete description of this sysctl.

*Availability: FreeBSD 9 or later.*

*New in version 3.4.*

resource. **RLIMIT_NPTS**

The maximum number of pseudo-terminals created by this user id.

*Availability: FreeBSD 9 or later.*

*New in version 3.4.*
Resource Usage

These functions are used to retrieve resource usage information:

resource. **getrusage**(who)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_\* constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields *ru_utime* and *ru_stime* of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage*(2) man page for detailed information about these values. A brief summary is presented here:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ru_utime</td>
<td>time in user mode (float)</td>
</tr>
<tr>
<td>1</td>
<td>ru_stime</td>
<td>time in system mode (float)</td>
</tr>
<tr>
<td>2</td>
<td>ru_maxrss</td>
<td>maximum resident set size</td>
</tr>
<tr>
<td>3</td>
<td>ru_ixrss</td>
<td>shared memory size</td>
</tr>
</tbody>
</table>
This function will raise a `ValueError` if an invalid `who` parameter is specified. It may also raise `error` exception in unusual circumstances.

**resource.** `getpagesize()`

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following `RUSAGE_*` symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

**resource.** `RUSAGE_SELF`

Pass to `getrusage()` to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.
**resource.** `RUSAGE_CHILDREN`

Pass to `getrusage()` to request resources consumed by child processes of the calling process which have been terminated and waited for.

**resource.** `RUSAGE_BOTH`

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.

**resource.** `RUSAGE_THREAD`

Pass to `getrusage()` to request resources consumed by the current thread. May not be available on all systems.

*New in version 3.2.*
nis — Interface to Sun’s NIS (Yellow Pages)

The nis module gives a thin wrapper around the NIS library, useful for central administration of several hosts.

Because NIS exists only on Unix systems, this module is only available for Unix.

The nis module defines the following functions:

nis.match(key, mapname, domain=default_domain)

Return the match for key in map mapname, or raise an error (nis.error) if there is none. Both should be strings, key is 8-bit clean. Return value is an arbitrary array of bytes (may contain NULL and other joys).

Note that mapname is first checked if it is an alias to another name.

The domain argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

nis.cat(mapname, domain=default_domain)

Return a dictionary mapping key to value such that match(key, mapname)==value. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

Note that mapname is first checked if it is an alias to another name.

The domain argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.
**nis.maps**(*domain=default_domain*)
Return a list of all valid maps.

The *domain* argument allows overriding the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

**nis.get_default_domain()**
Return the system default NIS domain.

The *nis* module defines the following exception:

*exception* **nis.error**
An error raised when a NIS function returns an error code.
syslog — Unix syslog library routines

This module provides an interface to the Unix syslog library routines. Refer to the Unix manual pages for a detailed description of the syslog facility.

This module wraps the system syslog family of routines. A pure Python library that can speak to a syslog server is available in the logging.handlers module as SysLogHandler.

The module defines the following functions:

syslog.syslog(message)

syslog.syslog(priority, message)

Send the string message to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a facility and a level. The optional priority argument, which defaults to LOG_INFO, determines the message priority. If the facility is not encoded in priority using logical-or (LOG_INFO | LOG_USER), the value given in the openlog() call is used.

If openlog() has not been called prior to the call to syslog(), openlog() will be called with no arguments.

syslog.openlog([ident[, logoption[, facility]]])

Logging options of subsequent syslog() calls can be set by calling openlog(). syslog() will call openlog() with no arguments if the log is not currently open.
The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional *facility* keyword argument (default is LOG_USER) sets the default facility for messages which do not have a facility explicitly encoded.

*Changed in version 3.2:* In previous versions, keyword arguments were not allowed, and *ident* was required. The default for *ident* was dependent on the system libraries, and often was `python` instead of the name of the Python program file.

**syslog.closelog()**
Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, `openlog()` will be called on the first `syslog()` call (if `openlog()` hasn’t already been called), and *ident* and other `openlog()` parameters are reset to defaults.

**syslog.setlogmask(maskpri)**
Set the priority mask to `maskpri` and return the previous mask value. Calls to `syslog()` with a priority level not set in `maskpri` are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

The module defines the following constants:

Priority levels (high to low):
LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

Facilities:
LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON, LOG_SYSLOG, LOG_LOCAL0 to LOG_LOCAL7, and, if defined in <syslog.h>, LOG_AUTHPRIV.

Log options:
LOG_PID, LOG_CONS, LOG_NDELAY, and, if defined in <syslog.h>, LOG_ODELAY, LOG_NOWAIT, and LOG_PERROR.
Examples

Simple example

A simple set of examples:

```python
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```python
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```
Superseded Modules

The modules described in this chapter are deprecated and only kept for backwards compatibility. They have been superseded by other modules.

- `optparse` — Parser for command line options
  - Background
    - Terminology
    - What are options for?
    - What are positional arguments for?
  - Tutorial
    - Understanding option actions
    - The store action
    - Handling boolean (flag) options
    - Other actions
    - Default values
    - Generating help
      - Grouping Options
    - Printing a version string
    - How `optparse` handles errors
    - Putting it all together
  - Reference Guide
    - Creating the parser
    - Populating the parser
    - Defining options
    - Option attributes
    - Standard option actions
    - Standard option types
    - Parsing arguments
    - Querying and manipulating your option parser
    - Conflicts between options
- Cleanup
- Other methods

  - **Option Callbacks**
    - Defining a callback option
    - How callbacks are called
    - Raising errors in a callback
    - Callback example 1: trivial callback
    - Callback example 2: check option order
    - Callback example 3: check option order (generalized)
    - Callback example 4: check arbitrary condition
    - Callback example 5: fixed arguments
    - Callback example 6: variable arguments

  - **Extending optparse**
    - Adding new types
    - Adding new actions

- `imp — Access the import internals`

- **Examples**
optparse — Parser for command line options

Source code: Lib/optparse.py

Deprecated since version 3.2: The optparse module is deprecated and will not be developed further; development will continue with the argparse module.

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old getopt module. optparse uses a more declarative style of command-line parsing: you create an instance of OptionParser, populate it with options, and parse the command line. optparse allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using optparse in a simple script:

```python
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                action="store_false", dest="verbose",
                help="don't print status messages")

(options, args) = parser.parse_args()
```
With these few lines of code, users of your script can now do the “usual thing” on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, `optparse` sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be "outfile" and `options.verbose` will be False. `optparse` supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -f outfile
<yourscript> -qf outfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and `optparse` will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
- -h, --help                  show this help message and exit
- -f FILE, --file=FILE        write report to FILE
- -q, --quiet                don't print status messages
```
where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).
Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argument

a string entered on the command-line, and passed by the shell to `exec1()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term “word”.

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read “argument” as “an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`”.

option

an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen (“-“) followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x` `-F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by
optparse.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren’t usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you’re exclusively targeting VMS, MS-DOS, and/or Windows.

**option argument**

An argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn’t. Lots
of people want an “optional option arguments” feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if \(-a\) takes an optional argument and \(-b\) is another option entirely, how do we interpret \(-ab\)? Because of this ambiguity, \texttt{optparse} does not support this feature.

Positional argument

something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

Required option

an option that must be supplied on the command-line; note that the phrase “required option” is self-contradictory in English. \texttt{optparse} doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

\begin{verbatim}
prog -v --report report.txt foo bar
\end{verbatim}

\texttt{-v} and \texttt{--report} are both options. Assuming that \texttt{--report} takes one argument, \texttt{report.txt} is an option argument. \texttt{foo} and \texttt{bar} are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn’t clear, options are usually \textit{optional}. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main
exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have “required options”. Think about it. If it's required, then it's not optional! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn’t make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in
order to run successfully, it doesn't much matter how you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the “Preferences” dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.
Tutorial

While optparse is quite flexible and powerful, it’s also straightforward to use in most cases. This section covers the code patterns that are common to any optparse-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```python
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```python
parser.add_option(opt_str, ..., attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell optparse what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```python
parser.add_option("-f", "--file", ...)
```

You’re free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to OptionParser.add_option() are effectively labels for the option defined by that call. For brevity, we will
frequently refer to *encountering an option* on the command line; in reality, `optparse` encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct `optparse` to parse your program's command line:

```python
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: `action`, `type`, `dest` (destination), and `help`. Of these, `action` is the most fundamental.

**Understanding option actions**

Actions tell `optparse` what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into `optparse`; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell `optparse` to store a value in some variable—for example, take a string from the command
line and store it in an attribute of options.

If you don't specify an option action, optparse defaults to store.

The store action

The most common option action is store, which tells optparse to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```python
def add_option(parser):
    parser.add_option("-f", "--file",
                       action="store", type="string",
                       dest="filename")
```

Now let's make up a fake command line and ask optparse to parse it:

```python
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When optparse sees the option string `-f`, it consumes the next argument, foo.txt, and stores it in options.filename. So, after this call to parse_args(), options.filename is "foo.txt".

Some other option types supported by optparse are int and float. Here's an option that expects an integer argument:

```python
def add_option(parser):
    parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly
acceptable. Also, there's no explicit action, since the default is store.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since -n42 (one argument) is equivalent to -n 42 (two arguments), the code

```python
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print 42.

If you don't specify a type, `optparse` assumes string. Combined with the fact that the default action is store, that means our first example can be a lot shorter:

```python
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, `optparse` figures out a sensible default from the option strings: if the first long option string is --foo-bar, then the default destination is foo_bar. If there are no long option strings, `optparse` looks at the first short option string: the default destination for -f is f.

`optparse` also includes the built-in `complex` type. Adding types is covered in section Extending `optparse`.

### Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. `optparse` supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:
Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When `optparse` encounters `-v` on the command line, it sets `options.verbose` to True; when it encounters `-q`, `options.verbose` is set to False.

Other actions

Some other actions supported by `optparse` are:

- "store_const"  
  store a constant value
- "append"  
  append this option's argument to a list
- "count"  
  increment a counter by one
- "callback"  
  call a specified function

These are covered in section Reference Guide, Reference Guide and section Option Callbacks.

Default values

All of the above examples involve setting some variable (the “destination”) when certain command-line options are seen. What
happens if those options are never seen? Since we didn’t supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. `optparse` lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want `optparse` to set `verbose` to `True` unless `-q` is seen, then we can do this:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the `destination` rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Consider this:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```python
parser.set_defaults(verbos=True)
```
As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

### Generating help

`optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a help value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```python
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
pheader.add_option("-v", "--verbose",
                  action="store_true", dest="verbose",
                  help="make lots of noise [default]

parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting"

parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output"

parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, in"
                  or expert [default: %default"
```

If `optparse` encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:
Usage: <yourscript> [options] arg1 arg2

Options:
- **h, --help** show this help message and exit
- **-v, --verbose** make lots of noise [default]
- **-q, --quiet** be vewwy quiet (I'm hunting)
- **-f FILE, --filename=FILE** write output to FILE
- **-m MODE, --mode=MODE** interaction mode: novice, intermediate, or expert [default: intermediate]

(If the help output is triggered by a help option, `optparse` exits after printing the help text.)

There's a lot going on here to help `optparse` generate the best possible help message:

- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

`optparse` expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, `optparse` uses a bland but sensible default: "Usage: %prog [options]", which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—`optparse` takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-
generated help message, e.g. for the “mode” option:

-\texttt{m \texttt{MODE}, \--mode=MODE}

Here, “\texttt{MODE}” is called the meta-variable: it stands for the argument that the user is expected to supply to \texttt{-m/\--mode}. By default, \texttt{optparse} converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that’s not what you want—for example, the \texttt{\--filename} option explicitly sets \texttt{metavar="FILE"}, resulting in this automatically-generated option description:

-\texttt{f \texttt{FILE}, \--filename=FILE}

This is important for more than just saving space, though: the manually written help text uses the meta-variable \texttt{FILE} to clue the user in that there’s a connection between the semi-formal syntax \texttt{-f \texttt{FILE}} and the informal semantic description “write output to \texttt{FILE}”. This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include \texttt{%default} in the help string—\texttt{optparse} will replace it with \texttt{str()} of the option’s default value. If an option has no default value (or the default value is \texttt{None}), \texttt{%default} expands to \texttt{none}.

**Grouping Options**

When dealing with many options, it is convenient to group these options for better help output. An \texttt{OptionParser} can contain several option groups, each of which can contain several options.
An option group is obtained using the class `OptionGroup`:

class optparse.**OptionGroup**(parser, title, description=None)

where

- parser is the `OptionParser` instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

`OptionGroup` inherits from `OptionContainer` (like `OptionParser`) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the `OptionParser` method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an `OptionGroup` to a parser is easy:

```python
group = OptionGroup(parser, "Dangerous Options",
    "Caution: use these options at your own risk. It is believed that some of them bite.

    group.add_option("-g", action="store_true", help="Group option.

    parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help                 show this help message and exit
  -v, --verbose             make lots of noise [default]
```
A bit more complete example might involve using more than one group: still extending the previous example:

```python
group = OptionGroup(parser, "Dangerous Options",
    "Caution: use these options at your own risk. It is believed that some of them bite.
    
    -g Group option.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

# Debug Options

# Debug options

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
    help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
    help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
- h, --help show this help message and exit
```

-Dangerous Options:

Caution: use these options at your own risk. It is believed that some of them bite.

- g Group option.

OptionGroup

A bit more complete example might involve using more than one group: still extending the previous example:

```python
group = OptionGroup(parser, "Dangerous Options",
    "Caution: use these options at your own risk. It is believed that some of them bite.
    
    -g Group option.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

# Debug Options

# Debug options

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
    help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
    help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```
-v, --verbose  make lots of noise [default]
-q, --quiet    be vewwy quiet (I'm hunting
-f FILE, --filename=FILE  write output to FILE
-m MODE, --mode=MODE  interaction mode: novice, int[
default: intermediate]

Dangerous Options:
   Caution: use these options at your own risk. It of them bite.

   -g               Group option.

Debug Options:
   -d, --debug      Print debug information
   -s, --sql        Print all SQL statements exe-
   -e               Print every action done

Another interesting method, in particular when working programmatical-
with option groups is:

OptionParser. get_option_group(opt_str)
   Return the OptionGroup to which the short or long option string
   opt_str (e.g. '-o' or '--option') belongs. If there's no such
   OptionGroup, return None.

Printing a version string

Similar to the brief usage string, optparse can also print a version
string for your program. You have to supply the string as the version
argument to OptionParser:

gparser = OptionParser(usage="%prog [-f] [-q]", versi
%prog is expanded just like it is in usage. Apart from that, version can contain anything you like. When you supply it, optparse automatically adds a --version option to your parser. If it encounters this option on the command line, it expands your version string (by replacing %prog), prints it to stdout, and exits.

For example, if your script is called /usr/bin/foo:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the version string:

OptionParser.print_version(file=None)

Print the version message for the current program (self.version) to file (default stdout). As with print_usage(), any occurrence of %prog in self.version is replaced with the name of the current program. Does nothing if self.version is empty or undefined.

OptionParser.get_version()

Same as print_version() but returns the version string instead of printing it.

How optparse handles errors

There are two broad classes of errors that optparse has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to OptionParser.add_option(), e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an
exception (either optparse.OptionError or TypeError) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. optparse can automatically detect some user errors, such as bad option arguments (passing -n 4x where -n takes an integer argument), missing arguments (-n at the end of the command line, where -n takes an argument of any type). Also, you can call OptionParser.error() to signal an application-defined error condition:

```python
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, optparse handles the error the same way: it prints the program’s usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes 4x to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]
foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]
```
foo: error: -n option requires an argument

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling OptionParser.error() from your application code.

If optparse's default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its exit() and/or error() methods.

Putting it all together

Here's what optparse-based scripts usually look like:

```python
from optparse import OptionParser
...

def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")

    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
```

main()
Reference Guide

Creating the parser

The first step in using `optparse` is to create an `OptionParser` instance.

```python
class optparse.OptionParser(...)
```

The `OptionParser` constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

```python
usage (default: "%prog [options]")
```

The usage summary to print when your program is run incorrectly or with a help option. When `optparse` prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

```python
option_list (default: [])
```

A list of `Option` objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by `OptionParser` subclasses), but before any version or help options. Deprecated; use `add_option()` after creating the parser instead.

```python
option_class (default: optparse.Option)
```

Class to use when adding options to the parser in `add_option()`.
version (default: None)

A version string to print when the user supplies a version option. If you supply a true value for version, optparse automatically adds a version option with the single option string --version. The substring %prog is expanded the same as for usage.

conflict_handler (default: "error")

Specifies what to do when options with conflicting option strings are added to the parser; see section Conflicts between options.

description (default: None)

A paragraph of text giving a brief overview of your program. optparse reformats this paragraph to fit the current terminal width and prints it when the user requests help (after usage, but before the list of options).

formatter (default: a new IndentedHelpFormatter)

An instance of optparse.HelpFormatter that will be used for printing help text. optparse provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (default: True)

If true, optparse will add a help option (with option strings -h and --help) to the parser.

prog

The string to use when expanding %prog in usage and version instead of os.path.basename(sys.argv[0]).

epilog (default: None)

A paragraph of help text to print after the option help.

Populating the parser
There are several ways to populate the parser with options. The preferred way is by using `OptionParser.add_option()`, as shown in section Tutorial. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```python
option_list = [
    make_option("-f", "--filename",
        action="store", type="string", dest=
    make_option("-q", "--quiet",
        action="store_false", dest="verbose"
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of `optparse` may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

### Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.
The canonical way to create an Option instance is with the `add_option()` method of `OptionParser`.

```python
OptionParser.add_option(option)
OptionParser.add_option(*opt_str, attr=value, ...)
```

To define an option with only a short option string:

```python
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```python
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `optparse` raises an OptionError exception explaining your mistake.

An option's `action` determines what `optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `optparse` are:

```
"store"
    store this option's argument (default)
"store_const"
    store a constant value
"store_true"
    store a true value
"store_false"
    store a false value
```
"append"
   append this option's argument to a list
"append_const"
   append a constant value to a list
"count"
   increment a counter by one
"callback"
   call a specified function
"help"
   print a usage message including all options and the documentation for them

(If you don't supply an action, the default is "store". For this action, you may also supply type and dest option attributes; see Standard option actions.)

As you can see, most actions involve storing or updating a value somewhere. optparse always creates a special object for this, conventionally called options (it happens to be an instance of optparse.Values). Option arguments (and various other values) are stored as attributes of this object, according to the dest (destination) option attribute.

For example, when you call

   parser.parse_args()

one of the first things optparse does is create the options object:

   options = Values()

If one of the options in this parser is defined with
and the command-line being parsed includes any of the following:

- `foo -f`  
- `f foo`  
- `--file=foo`  
- `--file foo`

then `optparse`, on seeing this option, will do the equivalent of

```python
options.filename = "foo"
```

The `type` and `dest` option attributes are almost as important as `action`, but `action` is the only one that makes sense for all options.

Option attributes

The following option attributes may be passed as keyword arguments to `OptionParser.add_option()`. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, `optparse` raises `OptionError`.

Option. `action`  
(default: "store")

Determines `optparse`'s behaviour when this option is seen on the command line; the available options are documented here.

Option. `type`  
(default: "string")
The argument type expected by this option (e.g., "string" or "int"); the available option types are documented here.

Option. **dest**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells `optparse` where to write it: `dest` names an attribute of the `options` object that `optparse` builds as it parses the command line.

Option. **default**

The value to use for this option's destination if the option is not seen on the command line. See also `OptionParser.set_defaults()`.

Option. **nargs**
(default: 1)

How many arguments of type `type` should be consumed when this option is seen. If > 1, `optparse` will store a tuple of values to `dest`.

Option. **const**

For actions that store a constant value, the constant value to store.

Option. **choices**

For options of type "choice", the list of strings the user may choose from.

Option. **callback**

For options with action "callback", the callable to call when this option is seen. See section `Option Callbacks` for detail on the
arguments passed to the callable.

Option. **callback_args**
Option. **callback_keywords**
Additional positional and keyword arguments to pass to *callback* after the four standard callback arguments.

Option. **help**
Help text to print for this option when listing all available options after the user supplies a *help* option (such as *--help*). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option. **metavar**
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section **Tutorial** for an example.

**Standard option actions**

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*’s behaviour; a few have required attributes, which you must specify for any option using that action.

- **"store"** [relevant: *type, dest, nargs, choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a
tuple. See the **Standard option types** section.

If `choices` is supplied (a list or tuple of strings), the type defaults to "choice".

If `type` is not supplied, it defaults to "string".

If `dest` is not supplied, `optparse` derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, `optparse` derives a destination from the first short option string (e.g., `-f` implies `f`).

Example:

```python
class parser:
    parser.add_option("-f")
    parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

`optparse` will set

```python
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [required: `const`; relevant: `dest`]

The value `const` is stored in `dest`.

Example:

```python
parser.add_option("-q", "--quiet",
```
If --noisy is seen, **optparse** will set

```python
options.verbose = 2
```

- **"store_true" [relevant: dest]**

  A special case of "store_const" that stores a true value to `dest`.

- **"store_false" [relevant: dest]**

  Like "store_true", but stores a false value.

  Example:

```python
parser.add_option("--clobber", action="store_true")
parser.add_option("--no-clobber", action="store_false")
```

- **"append" [relevant: type, dest, nargs, choices]**

  The option must be followed by an argument, which is appended to the list in `dest`. If no default value for `dest` is supplied, an empty list is automatically created when **optparse** first encounters this option on the command-line. If `nargs` > 1, multiple arguments are consumed, and a tuple of length `nargs` is appended to `dest`. 
The defaults for \texttt{type} and \texttt{dest} are the same as for the \texttt{"store"} action.

Example:

```python
defaults = {'type': 'store', 'dest': 'store'}
parser.add_option("-t", "--tracks", action="append",
                  dest='tracks',
                  help='specify a track')
```

If \texttt{-t3} is seen on the command-line, \texttt{optparse} does the equivalent of:

```python
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, \texttt{--tracks=4} is seen, it does:

```python
options.tracks.append(int("4"))
```

The \texttt{append} action calls the \texttt{append} method on the current value of the option. This means that any default value specified must have an \texttt{append} method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```python
>>> parser.add_option("--files", action="append")
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg',
>>> opts.files
["~/mypkg/defaults", 'overrides.mypkg']
```

- \texttt{"append\_const"} [required: \texttt{const}; relevant: \texttt{dest}]

Like \texttt{"store\_const"}, but the value \texttt{const} is appended to
dest; as with "append", dest defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: dest]

Increment the integer stored at dest. If no default value is supplied, dest is set to zero before being incremented the first time.

Example:

```python
define parser.add_option("-v", action="count", dest="verbosity")
```

The first time `-v` is seen on the command line, optparse does the equivalent of:

```python
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```python
options.verbosity += 1
```

- "callback" [required: callback; relevant: type, nargs, callback_args, callback_kwargs]

Call the function specified by callback, which is called as

```python
func(option, opt_str, value, parser, *args, **kwargs)
```

See section Option Callbacks for more detail.
- "help"

Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to OptionParser's constructor and the help string passed to every option.

If no help string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

optparse automatically adds a help option to all OptionParsers, so you do not normally need to create one.

Example:

```python
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, # be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="help", help="Be moderately verbose")
parser.add_option("--file", dest="filename", help="Input file to read data")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If optparse sees either -h or --help on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

```
Usage: foo.py [options]
```
Options:
- `h`, `--help`    Show this help message and exit
- `v`             Be moderately verbose
- `--file=FILENAME` Input file to read data from

After printing the help message, `optparse` terminates your process with `sys.exit(0)`.

- "version"

Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with `help` options, you will rarely create `version` options, since `optparse` automatically adds them when needed.

Standard option types

`optparse` has five built-in option types: "string", "int", "choice", "float" and "complex". If you need to add new option types, see section Extending `optparse`.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type "int") are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling `int()` with the appropriate base (2, 8, 10, or 16). If this fails, so will `optparse`, although with a more useful error message.

"float" and "complex" option arguments are converted directly with `float()` and `complex()`, with similar error-handling.

"choice" options are a subtype of "string" options. The `choices` option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

**Parsing arguments**

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```python
(options, args) = parser.parse_args(args=None, values=optparse.Values())
```

where the input parameters are

- `args` the list of arguments to process (default: `sys.argv[1:]`)
- `values` an `optparse.Values` object to store option arguments in (default: a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it
and the return values are

```python
options
    the same object that was passed in as values, or the optparse.Values instance created by optparse

args
    the leftover positional arguments after all options have been processed
```

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated `setattr()` calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

### Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

**OptionParser**. `disable_interspersed_args()`  
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, `optparse` normally accepts this syntax:

```
prog -a arg1 -b arg2
```
and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call `disable_interspersed_args()`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don’t get confused. For example, each command might have a different set of options.

**OptionParser.enable_interspersed_args()**

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

**OptionParser.get_option(opt_str)**

Returns the Option instance with the option string `opt_str`, or None if no options have that option string.

**OptionParser.has_option(opt_str)**

Return true if the OptionParser has an option with option string `opt_str` (e.g., `-q` or `--verbose`).

**OptionParser.remove_option(opt_str)**

If the OptionParser has an option corresponding to `opt_str`, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If `opt_str` does not occur in any option belonging to this OptionParser, raises `ValueError`. 
Conflicts between options

If you’re not careful, it’s easy to define options with conflicting option strings:

```python
parser.add_option("-n", "--dry-run", ...) 
... 
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you’ve defined your own OptionParser subclass with some standard options.)

Every time you add an option, `optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```python
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```python
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

- "error" (default)
  assume option conflicts are a programming error and raise `OptionConflictError`
- "resolve"
  resolve option conflicts intelligently (see below)

As an example, let’s define an `OptionParser` that resolves conflicts
intelligently and add conflicting options to it:

```python
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm"
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, `optparse` detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is "resolve", it resolves the situation by removing `-n` from the earlier option’s list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
- dry-run do no harm
  ...
- n, --noisy be noisy
```

It’s possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `optparse` removes that option completely, so it doesn’t show up in help text or anywhere else. Carrying on with our existing OptionParser:

```python
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n/--dry-run` option is no longer accessible, so `optparse` removes it, leaving this help text:

```
Options:
  ...
- n, --noisy be noisy
```
Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling destroy() on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

OptionParser.set_usage(usage)

Set the usage string according to the rules described above for the usage constructor keyword argument. Passing None sets the default usage string; use optparse.SUPPRESS_USAGE to suppress a usage message.

OptionParser.print_usage(file=None)

Print the usage message for the current program (self.usage) to file (default stdout). Any occurrence of the string %prog in self.usage is replaced with the name of the current program. Does nothing if self.usage is empty or not defined.

OptionParser.get_usage()

Same as print_usage() but returns the usage string instead of printing it.
OptionParser.set_defaults(dest=value,...)

Set default values for several option destinations at once. Using set_defaults() is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several “mode” options all set the same destination, any one of them can set the default, and the last one wins:

```python
parser.add_option("--advanced", action="store_const", 
                 dest="mode", const="advanced", 
                 default="novice")  # overrides

parser.add_option("--novice", action="store_const", 
                 dest="mode", const="novice", 
                 default="advanced")  # override
```

To avoid this confusion, use set_defaults():

```python
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const", 
                 dest="mode", const="advanced")
parser.add_option("--novice", action="store_const", 
                 dest="mode", const="novice")
```
Option Callbacks

When `optparse`'s built-in actions and types aren’t quite enough for your needs, you have two choices: extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the "callback" action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call:

```python
parser.add_option("-c", action="callback", callback=
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn’t even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this
section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via callback_args and callback_kwargs. Thus, the minimal callback function signature is:

```python
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

**type**

has its usual meaning: as with the "store" or "append" actions, it instructs optparse to consume one argument and convert it to type. Rather than storing the converted value(s) anywhere, though, optparse passes it to your callback function.

**nargs**

also has its usual meaning: if it is supplied and > 1, optparse will consume nargs arguments, each of which must be convertible to type. It then passes a tuple of converted values to your callback.

**callback_args**

a tuple of extra positional arguments to pass to the callback

**callback_kwargs**

a dictionary of extra keyword arguments to pass to the callback

### How callbacks are called

All callbacks are called as follows:
```python
func(option, opt_str, value, parser, *args, **kwargs)
```

where

- **option**
  is the Option instance that's calling the callback

- **opt_str**
  is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

- **value**
  is the argument to this option seen on the command-line. `optparse` will only expect an argument if `type` is set; the type of value will be the type implied by the option's type. If `type` for this option is `None` (no argument expected), then `value` will be `None`. If `nargs > 1`, `value` will be a tuple of values of the appropriate type.

- **parser**
  is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

  - **parser.largs**
    the current list of leftover arguments, i.e. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

  - **parser.rargs**
the current list of remaining arguments, i.e. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

`parser.values`:

the object where option values are by default stored (an instance of `optparse.OptionValues`). This lets callbacks use the same mechanism as the rest of `optparse` for storing option values; you don’t need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

`args`:

is a tuple of arbitrary positional arguments supplied via the `callback_args` option attribute.

`kwargs`:

is a dictionary of arbitrary keyword arguments supplied via `callback_kwargs`.

**Raising errors in a callback**

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). `optparse` catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

**Callback example 1: trivial callback**

Here’s an example of a callback option that takes no arguments, and simply records that the option was seen:
```python
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True
```
Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```python
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full" % opt_str)
    setattr(parser.values, option.dest, 1)

parser.add_option("--foo",
    action="callback", callback=check_moon)
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define `type`, then the option takes one argument that must be convertible to that type; if you further define `nargs`, then the option takes `nargs` arguments.
Here’s an example that just emulates the standard "store" action:

```python
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
    action="callback", callback=store_value,
    type="int", nargs=3, dest="foo")
```

Note that `optparse` takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don’t need a callback for this example.)

**Callback example 6: variable arguments**

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as `optparse` doesn’t provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that `optparse` normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you’re willing to make for your application (which is why `optparse` doesn’t
support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```python
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

def floatable(str):
    try:
        float(str)
        return True
    except ValueError:
        return False

for arg in parser.rargs:
    # stop on --foo like options
    if arg[:2] == "--" and len(arg) > 2:
        break
    # stop on -a, but not on -3 or -3.0
    if arg[:1] == "-" and len(arg) > 1 and not floatable:
        break
    value.append(arg)

del parser.rargs[:len(value)]
setattr(parser.values, option.dest, value)
...
parser.add_option("-c", "--callback", dest="vararg_attr",
    action="callback", callback=vararg_callback)
```
Extending `optparse`

Since the two major controlling factors in how `optparse` interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

**Adding new types**

To add new types, you need to define your own subclass of `optparse`'s `Option` class. This class has a couple of attributes that define `optparse`'s types: `TYPES` and `TYPE_CHECKER`.

**Option. `TYPES`**

A tuple of type names; in your subclass, simply define a new tuple `TYPES` that builds on the standard one.

**Option. `TYPE_CHECKER`**

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```python
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type mytype. The value returned by a type-checking function will wind up in the `OptionValues` instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.
Your type-checking function should raise OptionValueError if it encounters any problems. OptionValueError takes a single string argument, which is passed as-is to OptionParser's error() method, which in turn prepends the program name and the string "error:" and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a "complex" option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because optparse 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```python
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError("option %s: invalid complex value: %r").format(opt, value)
```

Finally, the Option subclass:

```python
class MyOption(Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```
(If we didn’t make a copy() of `Option.TYPE_CHECKER`, we would end up modifying the `TYPE_CHECKER` attribute of `optparse`’s `Option` class. This being Python, nothing stops you from doing that except good manners and common sense.)

That’s it! Now you can write a script that uses the new option type just like any other `optparse`-based script, except you have to instruct your `OptionParser` to use `MyOption` instead of `Option`:

```python
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to `OptionParser`; if you don’t use `add_option()` in the above way, you don’t need to tell `OptionParser` which option class to use:

```python
option_list = [MyOption("-c", action="store", type='
parser = OptionParser(option_list=option_list)
```

### Adding new actions

Adding new actions is a bit trickier, because you have to understand that `optparse` has a couple of classifications for actions:

**“store” actions**
- actions that result in `optparse` storing a value to an attribute of the current `OptionValues` instance; these options require a `dest` attribute to be supplied to the `Option` constructor.

**“typed” actions**
- actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `type` attribute to the `Option` constructor.
These are overlapping sets: some default "store" actions are "store", "store_const", "append", and "count", while the default “typed” actions are "store", "append", and "callback".

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

Option. **ACTIONS**

All actions must be listed in ACTIONS.

Option. **STORE_ACTIONS**

"store" actions are additionally listed here.

Option. **TYPED_ACTIONS**

"typed" actions are additionally listed here.

Option. **ALWAYS_TYPED_ACTIONS**

Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that `optparse` assigns the default type, "string", to options with no explicit type whose action is listed in **ALWAYS_TYPED_ACTIONS**.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an
existing list with them. That is, if `--names` is an "extend" option of type "string", the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```python
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend")
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend")
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS

    def take_action(self, action, dest, opt, value, ...
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(...
```

Features of note:

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both `STORE_ACTIONS` and `TYPED_ACTIONS`.

- to ensure that `optparse` assigns the default type of "string" to "extend" actions, we put the "extend" action in
AlwaysTypedActions as well.

- MyOption.take_action() implements just this one new action, and passes control back to Option.take_action() for the standard optparse actions.

- values is an instance of the optparse_parser.Values class, which provides the very useful ensure_value() method. ensure_value() is essentially getattr() with a safety valve; it is called as

```python
values.ensure_value(attr, value)
```

If the attr attribute of values doesn’t exist or is None, then ensure_value() first sets it to value, and then returns ‘value. This is very handy for actions like "extend", "append", and "count", all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using ensure_value() means that scripts using your action don’t have to worry about setting a default value for the option destinations in question; they can just leave the default as None and ensure_value() will take care of getting it right when it’s needed.
imp — Access the import internals

Source code: Lib/imp.py

Deprecated since version 3.4: The imp package is pending deprecation in favor of importlib.

This module provides an interface to the mechanisms used to implement the import statement. It defines the following constants and functions:

imp.get_magic()
  Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

  Deprecated since version 3.4: Use importlib.util.MAGIC_NUMBER instead.

imp.get_suffixes()
  Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in open() function to open the file (this can be 'r' for text files or 'rb' for binary files), and type is the file type, which has one of the values PY_SOURCE, PY_COMPILED, or C_EXTENSION, described below.

  Deprecated since version 3.3: Use the constants defined on
**importlib.machinery** instead.

**imp. find_module(name[, path])**

Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (**C_BUILTIN**), then a frozen module (**PY_FROZEN**), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by `get_suffixes()` above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple (`file`, `pathname`, `description`):

*file* is an open *file object* positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by `get_suffixes()` describing the kind of module found.

If the module does not live in a file, the returned *file* is `None`, *pathname* is the empty string, and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is `None`, *pathname* is the package
path and the last item in the description tuple is `PKG_DIRECTORY`.

This function does not handle hierarchical module names (names containing dots). In order to find `P.M`, that is, submodule `M` of package `P`, use `find_module()` and `load_module()` to find and load package `P`, and then use `find_module()` with the path argument set to `P.__path__`. When `P` itself has a dotted name, apply this recipe recursively.

*Deprecated since version 3.3:* Use `importlib.util.find_spec()` instead unless Python 3.3 compatibility is required, in which case use `importlib.find_loader()`. For example usage of the former case, see the Examples section of the `importlib` documentation.

```python
imp.load_module(name, file, pathname, description)
```

Load a module that was previously found by `find_module()` (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it will reload the module! The name argument indicates the full module name (including the package name, if this is a submodule of a package). The file argument is an open file, and pathname is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The description argument is a tuple, as would be returned by `get_suffixes()`, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually `ImportError`) is raised.

**Important:** the caller is responsible for closing the file argument, if
it was not None, even when an exception is raised. This is best done using a try ... finally statement.

*Deprecated since version 3.3:* If previously used in conjunction with `imp.find_module()` then consider using `importlib.import_module()`, otherwise use the loader returned by the replacement you chose for `imp.find_module()`. If you called `imp.load_module()` and related functions directly with file path arguments then use a combination of `importlib.util.spec_from_file_location()` and `importlib.util.module_from_spec()`. See the Examples section of the `importlib` documentation for details of the various approaches.

```
imp.new_module(name)
```

Return a new empty module object called `name`. This object is not inserted in `sys.modules`.

*Deprecated since version 3.4:* Use `importlib.util.module_from_spec()` instead.

```
imp.reload(module)
```

Reload a previously imported `module`. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the `module` argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level
code reexecuted, defining a new set of objects which are bound to names in the module’s dictionary. The init function of extension modules is not called a second time.

- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a try statement it can test for the table’s presence and skip its initialization if desired:

```python
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for sys, __main__ and builtins. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.
If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (`module.*name*`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

*Changed in version 3.3:* Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

*Deprecated since version 3.4:* Use `importlib.reload()` instead.

The following functions are conveniences for handling PEP 3147 byte-compiled file paths.

*New in version 3.2.*

```python
imp.cache_from_source(path, debug_override=None)
```

Return the PEP 3147 path to the byte-compiled file associated with the source `path`. For example, if `path` is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`; if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised). By passing in `True` or `False` for `debug_override` you can override the system's value for...
__debug__, leading to optimized bytecode.

*path* need not exist.

**Changed in version 3.3:** If `sys.implementation.cache_tag` is None, then `NotImplementedError` is raised.

**Deprecated since version 3.4:** Use `importlib.util.cache_from_source()` instead.

**Changed in version 3.5:** The `debug_override` parameter no longer creates a `.pyo` file.

```python
imp.source_from_cache(path)
```

Given the *path* to a [PEP 3147](https://www.python.org/dev/peps/pep-3147/) file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to [PEP 3147](https://www.python.org/dev/peps/pep-3147/) format, a `ValueError` is raised. If `sys.implementation.cache_tag` is not defined, `NotImplementedError` is raised.

**Changed in version 3.3:** Raise `NotImplementedError` when `sys.implementation.cache_tag` is not defined.

**Deprecated since version 3.4:** Use `importlib.util.source_from_cache()` instead.

```python
imp.get_tag()
```

Return the [PEP 3147](https://www.python.org/dev/peps/pep-3147/) magic tag string matching this version of Python's magic number, as returned by `get_magic()`.

**Deprecated since version 3.4:** Use

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

**imp.lock_held()**

Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

*Changed in version 3.3:* The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

*Deprecated since version 3.4.*

**imp.acquire_lock()**

Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.
Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

*Changed in version 3.3:* The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

*Deprecated since version 3.4.*

```
imp.release_lock()
```

Release the interpreter’s global import lock. On platforms without threads, this function does nothing.

*Changed in version 3.3:* The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

*Deprecated since version 3.4.*

The following constants with integer values, defined in this module, are used to indicate the search result of `find_module()`.

```
imp.PY_SOURCE
```

The module was found as a source file.

*Deprecated since version 3.3.*

```
imp.PY_COMPILED
```

The module was found as a compiled code object file.

*Deprecated since version 3.3.*
imp.C_EXTENSION
The module was found as dynamically loadable shared library.

_Deprecated since version 3.3._

imp.PKG DIRECTORY
The module was found as a package directory.

_Deprecated since version 3.3._

imp.C_BUILTIN
The module was found as a built-in module.

_Deprecated since version 3.3._

imp.PY_FROZEN
The module was found as a frozen module.

_Deprecated since version 3.3._

class imp.NullImporter(path_string)
The NullImporter type is a _PEP 302_ import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises _ImportError_. Otherwise, a NullImporter instance is returned.

Instances have only one method:

find_module(fullname[, path])
This method always returns None, indicating that the requested module could not be found.

Changed in version 3.3: None is inserted into sys.path_importer_cache instead of an instance of
NullImporter.

*Deprecated since version 3.4:* Insert None into `sys.path_importer_cache` instead.
Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This implementation wouldn’t work in that version, since $\text{find\_module()}$ has been extended and $\text{load\_module()}$ has been added in 1.4.)

```python
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=[]):
    # Fast path: see if the module has already been loaded.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception, there's a problem we can't handle -- let the caller handle it.
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp.
        if fp:
            fp.close()
```

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Undocumented Modules

Here’s a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.
Platform specific modules

These modules are used to implement the `os.path` module, and are not documented beyond this mention. There's little need to document these.

`ntpath`
— Implementation of `os.path` on Win32 and Win64 platforms.

`posixpath`
— Implementation of `os.path` on POSIX.
1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such extension modules can do two things that can’t be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h".

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

**Note:** The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the `ctypes` module or the `cffi` library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.
1.1. A Simple Example

Let's create an extension module called spam (the favorite food of Monty Python fans...) and let's say we want to create a Python interface to the C library function system() [1]. This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file spammodule.c. (Historically, if a module is called spam, the C file containing its implementation is called spammodule.c; if the module name is very long, like spammify, the module name can be just spammify.c.)

The first line of our file can be:

```
#include <Python.h>
```

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

**Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include Python.h before any standard headers are included.

All user-visible symbols defined by Python.h have a prefix of Py or PY, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: <stdio.h>,
<string.h>, <errno.h>, and <stdlib.h>. If the latter header file does not exist on your system, it declares the functions malloc(), free() and realloc() directly.

The next thing we add to our module file is the C function that will be called when the Python expression spam.system(string) is evaluated (we'll see shortly how it ends up being called):

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression "ls -l") to the arguments passed to the C function. The C function always has two arguments, conventionally named self and args.

The **self** argument points to the module object for module-level functions; for a method it would point to the object instance.

The **args** argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function PyArg_ParseTuple() in the Python API checks the argument types and converts them to C values. It uses a
template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return `NULL` immediately (as we saw in the example).
1.2. Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a NULL pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is NULL no exception has occurred. A second global variable stores the “associated value” of the exception (the second argument to `raise`). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the result in Python of `sys.exc_info()` (see the section on module `sys` in the Python Library Reference). It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the “associated value” of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don’t need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with
`PyErr_Occurred()`. This returns the current exception object, or NULL if no exception has occurred. You normally don't need to call `PyErr_Occurred()` to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function \( f \) that calls another function \( g \) detects that the latter fails, \( f \) should itself return an error value (usually NULL or -1). It should not call one of the `PyErr_*()` functions — one has already been called by \( g \). \( f \)'s caller is then supposed to also return an error indication to its caller, again without calling `PyErr_*()`, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter’s main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*()` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling `PyErr_Clear()`. The only time C code should call `PyErr_Clear()` is if it doesn’t want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

Every failing `malloc()` call must be turned into an exception — the direct caller of `malloc()` (or `realloc()`) must call `PyErr_NoMemory()` and return a failure indicator itself. All the
object-creating functions (for example, `PyLong_FromLong()` already do this, so this note is only relevant to those who call `malloc()` directly.

Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and -1 for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don’t use `PyExc_TypeError` to mean that a file couldn’t be opened (that should probably be `PyExc_IOError`). If something’s wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

```c
static PyObject *SpamError;
```

and initialize it in your module’s initialization function (`PyInit_spam()`) with an exception object (leaving out the error
checking for now):

```python
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
    return m;
}
```

Note that the Python name for the exception object is spam.error. The `PyErr_NewException()` function may create a class with the base class being `Exception` (unless another class is passed in instead of `NULL`), described in Built-in Exceptions.

Note also that the SpamError variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing SpamError to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of `PyMODINIT_FUNC` as a function return type later in this sample.

The spam.error exception can be raised in your extension module
using a call to `PyErr_SetString()` as shown below:

```c
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed"用车
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```
1.3. Back to the Example

Going back to our example function, you should now be able to understand this statement:

```c
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns `NULL` (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as `const char *command`).

The next statement is a call to the Unix function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```c
sts = system(command);
```

Our spam.\texttt{system()} function must return the value of `sts` as a Python object. This is done using the function `PyLong_FromLong()`:

```c
return PyLong_FromLong(sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning \texttt{void}), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the
Py_RETURN_NONE macro):

```c
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a `NULL` pointer, which means “error” in most contexts, as we have seen.
1.4. The Module’s Method Table and Initialization Function

I promised to show how spam_system() is called from Python programs. First, we need to list its name and address in a “method table”:

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

Note the third entry (METH_VARARGS). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be METH_VARARGS or METH_VARARGS | METH_KEYWORDS; a value of 0 means that an obsolete variant of PyArg_ParseTuple() is used.

When using only METH_VARARGS, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via PyArg_ParseTuple(); more information on this function is provided below.

The METH_KEYWORDS bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third PyObject * parameter which will be a dictionary of keywords. Use PyArg_ParseTupleAndKeywords() to parse the arguments to such a function.
The method table must be referenced in the module definition structure:

```c
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam", /* name of module */
    spam_doc, /* module documentation, may be NULL */
    -1, /* size of per-interpreter state of the module, or -1 if the module keeps state in global variables. */
    SpamMethods
};
```

This structure, in turn, must be passed to the interpreter in the module's initialization function. The initialization function must be named `PyInit_name()`, where `name` is the name of the module, and should be the only non-static item defined in the module file:

```c
PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
```

Note that `PyMODINIT_FUNC` declares the function as `PyObject *` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

When the Python program imports module `spam` for the first time, `PyInit_spam()` is called. (See below for comments about embedding Python.) It calls `PyModule_Create()`, which returns a module object, and inserts built-in function objects into the newly created module based upon the table (an array of `PyMethodDef` structures) found in the module definition. `PyModule_Create()`
returns a pointer to the module object that it creates. It may abort with a fatal error for certain errors, or return NULL if the module could not be initialized satisfactorily. The init function must return the module object to its caller, so that it then gets inserted into `sys.modules`.

When embedding Python, the `PyInit_spam()` function is not called automatically unless there's an entry in the `PyImport_Inittab` table. To add the module to the initialization table, use `PyImport_AppendInittab()`, optionally followed by an import of the module:

```c
int main(int argc, char *argv[]) {
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    PyImport_AppendInittab("spam", PyInit_spam);

    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(program);

    /* Initialize the Python interpreter. Required. */
    Py_Initialize();

    /* Optionally import the module; alternatively, import can be deferred until the embedded script imports it. */
    PyImport_ImportModule("spam");

    ...
PyMem_RawFree(program);
return 0;

Note: Removing entries from sys.modules or importing compiled modules into multiple interpreters within a process (or following a fork() without an intervening exec()) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures.

A more substantial example module is included in the Python source distribution as Modules/xxmodule.c. This file may be used as a template or simply read as an example.

Note: Unlike our spam example, xxmodule uses multi-phase initialization (new in Python 3.5), where a PyModuleDef structure is returned from PyInit_spam, and creation of the module is left to the import machinery. For details on multi-phase initialization, see PEP 489.
1.5. Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter Building C and C++ Extensions) and additional information that pertains only to building on Windows (chapter Building C and C++ Extensions on Windows) for more information about this.

If you can’t use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (spammodule.c for example) in the Modules/ directory of an unpacked source distribution, add a line to the file Modules/Setup.local describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running `make` in the toplevel directory. You can also run `make` in the Modules/ subdirectory, but then you must first rebuild Makefile there by running `make Makefile`. (This is necessary each time you change the Setup file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```
1.6. Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called “callback” functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won’t dwell on how to call the Python parser with a particular string as input — if you’re interested, have a look at the implementation of the -c command line option in Modules/main.c from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to Py_INCREF() it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```c
static PyObject *my_callback = NULL;

static PyObject *my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp
```
if (!PyCallable_Check(temp)) {
    PyErr_SetString(PyExc_TypeError, "parameter must be callable");
    return NULL;
}
Py_XINCREF(temp);    /* Add a reference */
Py_XDECREF(my_callback);   /* Dispose of prev
my_callback = temp;      /* Remember new call */
/* Boilerplate to return "None" */
Py_INCREF(Py_None);
result = Py_None;
}
return result;

This function must be registered with the interpreter using the METH_VARARGS flag; this is described in section The Module's Method Table and Initialization Function. The PyArg_ParseTuple() function and its arguments are documented in section Extracting Parameters in Extension Functions.

The macros Py_XINCREF() and Py_XDECREF() increment/decrement the reference count of an object and are safe in the presence of NULL pointers (but note that temp will not be NULL in this context). More info on them in section Reference Counts.

Later, when it is time to call the function, you call the C function PyObject_CallObject(). This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in NULL, or an empty tuple; to call it with one argument, pass a singleton tuple. Py_BuildValue() returns a tuple when its format string consists of zero or more format codes between
parentheses. For example:

```c
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` returns a Python object pointer: this is the return value of the Python function. `PyObject_CallObject()` is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t `NULL`. If it is, the Python function terminated by raising an exception. If the C code that called `PyObject_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling
PyErr_Clear(). For example:

```c
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```c
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword
arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```python
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_XDECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_XDECREF(result);
```
1.7. Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

```c
int PyArg_ParseTuple(PyObject *arg, const char *format, ...) {
```

The `arg` argument must be a tuple object containing an argument list passed from Python to a C function. The `format` argument must be a format string, whose syntax is explained in Parsing arguments and building values in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are borrowed references; do not decrement their reference count!

Some example calls:

```c
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t */
#include <Python.h>

int ok;
int i, j;
```
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, """); /* No arguments */
/* Python call: f() */

ok = PyArg_ParseTuple(args, "s", &s); /* A string */
/* Possible Python call: f('whoops!') */

ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
/* Possible Python call: f(1, 2, 'three') */

ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
/* A pair of ints and a string, whose size is also returned */
/* Possible Python call: f((1, 2), 'three') */

{ const char *file;
  const char *mode = "r";
  int bufsize = 0;
  ok = PyArg_ParseTuple(args, "s|si", &file, &mode);
  /* A string, and optionally another string and an integer */
  /* Possible Python calls: 
     f('spam')
     f('spam', 'w')
     f('spam', 'wb', 1000000) */
}

{ int left, top, right, bottom, h, v;
  ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
                        &left, &top, &right, &bottom, &h, &v);
/* A rectangle and a point */
/* Possible Python call: */
f(((0, 0), (400, 300)), (10, 10)) */
}

{ }

Py_complex c;
ok = PyArg_ParseTuple(args, "D:myfunction", &c);
/* a complex, also providing a function name for */
/* Possible Python call: myfunction(1+2j) */
}
1.8. Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```c
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject const char *format,
                                 PyObject *kwdict, PyObject *kwlist)
```

The `arg` and `format` parameters are identical to those of the `PyArg_ParseTuple()` function. The `kwdict` parameter is the dictionary of keywords received as the third parameter from the Python runtime. The `kwlist` parameter is a `NULL`-terminated list of strings which identify the parameters; the names are matched with the type information from `format` from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

**Note:** Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the `kwlist` will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```c
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *kwdict)
{
    int voltage;
    ```
const char *state = "a stiff";
const char *action = "voom";
const char *type = "Norwegian Blue";

static char *kwlist[] = {"voltage", "state", "action"};

if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", &voltage, &state, &action))
    return NULL;

printf("-- This parrot wouldn't %s if you put %i Volts through it.
" action, voltage);
printf("-- Lovely plumage, the %s -- It's %s!\n" type);

Py_RETURN_NONE;

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values only take two PyObject* parameters, and keywdarg_parrot() takes three. */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS,
        "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
return PyModule_Create(&keywdargmodule);
1.9. Building Arbitrary Values

This function is the counterpart to *PyArg_ParseTuple*. It is declared as follows:

```c
PyObject *Py_BuildValue(const char *format, ...);
```

It recognizes a set of format units similar to the ones recognized by *PyArg_ParseTuple*, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with *PyArg_ParseTuple*: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), *Py_BuildValue* does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

```
Py_BuildValue(""")                 None
Py_BuildValue("i", 123)            123
Py_BuildValue("iii", 123, 456, 789) (123, 456,
Py_BuildValue("s", "hello")        'hello'
Py_BuildValue("y", "hello")        b'hello'
Py_BuildValue("ss", "hello", "world") ('hello', 'world')
Py_BuildValue("s#", "hello", 4)     'hell'    # One pointer
Py_BuildValue("y#", "hello", 4)     b'hell'
Py_BuildValue("()")                ()
```
Py_BuildValue("(i)", 123)    (123,)
Py_BuildValue("(ii)", 123, 456) (123, 456)
Py_BuildValue("(i,i)", 123, 456) (123, 456)
Py_BuildValue("[i,i]", 123, 456) [123, 456]
Py_BuildValue("{s:i,s:i}",
    "abc", 123, "def", 456) {'abc': 123
Py_BuildValue("((ii)(ii)) (ii)",
    1, 2, 3, 4, 5, 6) (((1, 2), (3, 4)), (5, 6))
1.10. Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a memory leak. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called using freed memory. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It’s easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it’s important to prevent leaks from
happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of malloc() and free(), it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called reference counting. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called automatic garbage collection. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of “automatic” to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn’t need to call free() explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions malloc() and free() are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we’ll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-
zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them. The gc module exposes a way to run the detector (the collect() function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the --without-cycle-gc option to the configure script on Unix platforms (including Mac OS X). If the cycle detector is disabled in this way, the gc module will not be available.

1.10.1. Reference Counting in Python

There are two macros, Py_INCREF(x) and Py_DECREF(x), which handle the incrementing and decrementing of the reference count. Py_DECREF() also frees the object when the count reaches zero. For flexibility, it doesn’t call free() directly — rather, it makes a call through a function pointer in the object’s type object. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use Py_INCREF(x) and Py_DECREF(x)? Let’s first introduce some terms. Nobody “owns” an object; however, you can own a reference to an object. An object’s reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling Py_DECREF() when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned
reference: pass it on, store it, or call \texttt{Py\_DECREF()}. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to \textit{borrow} \cite{2} a reference to an object. The borrower of a reference should not call \texttt{Py\_DECREF()}. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely \cite{3}.

The advantage of borrowing over owning a reference is that you don’t need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don’t run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling \texttt{Py\_INCREF()}. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

\textbf{1.10.2. Ownership Rules}

Whenever an object reference is passed into or out of a function, it is part of the function’s interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to
create a new object, such as `PyLong_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyLong_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them — even if they fail! (Note that `PyDict_SetItem()` and friends don’t take over ownership — they are “normal.”)

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference’s lifetime is guaranteed until the function
returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling \texttt{Py\_INCREF()}.

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred from the function to its caller.

1.10.3. Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using \texttt{Py\_DECREF()} on an unrelated object while borrowing a reference to a list item. For instance:

```c
void
bug(PyObject *list)
{
    PyObject *item = PyList\_GetItem(list, 0);
    PyList\_SetItem(list, 1, PyLong\_FromLong(0L));
    PyObject\_Print(item, stdout, 0);  /* BUG! */
}
```

This function first borrows a reference to \texttt{list[0]}, then replaces \texttt{list[1]} with the value \texttt{0}, and finally prints the borrowed reference. Looks harmless, right? But it’s not!

Let’s follow the control flow into \texttt{PyList\_SetItem()}. The list owns
references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a \_\_del\__( ) method. If this class instance has a reference count of 1, disposing of it will call its \_\_del\__( ) method.

Since it is written in Python, the \_\_del\__( ) method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to \texttt{item} in \texttt{bug()}? You bet! Assuming that the list passed into \texttt{bug()} is accessible to the \_\_del\__( ) method, it could execute a statement to the effect of \texttt{del list[0]}, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating \texttt{item}.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```c
void no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREFF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his \_\_del\__( ) methods would fail...

The second case of problems with a borrowed reference is a variant
involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```c
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

### 1.10.4. NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them `NULL` pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return `NULL` only to indicate that an exception occurred. The reason for not testing for `NULL` arguments is that functions often pass the objects they receive on to other function — if each function were to test for `NULL`, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for `NULL` only at the “source:” when a pointer that may be `NULL` is received, for example, from `malloc()` or from a function that may raise an exception.
The macros `Py_INCREF()` and `Py_DECREF()` do not check for `NULL` pointers — however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`PyType_Check()`) don’t check for `NULL` pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with `NULL` checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never `NULL` — in fact it guarantees that it is always a tuple [4].

It is a severe error to ever let a `NULL` pointer “escape” to the Python user.
1.11. Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).
1.12. Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type “collection” which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them static, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared static, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section The Module's Method Table and Initialization).
Function). And it means that symbols that should be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (void *). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module’s namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

Whichever method you choose, it’s important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (const char *); you’re permitted to pass in a NULL name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule’s name
matches this convention. This behavior gives C API users a high
degree of certainty that the Capsule they load contains the correct C
API.

The following example demonstrates an approach that puts most of
the burden on the writer of the exporting module, which is appropriate
for commonly used library modules. It stores all C API pointers (just
one in the example!) in an array of void pointers which becomes the
value of a Capsule. The header file corresponding to the module
provides a macro that takes care of importing the module and
retrieving its C API pointers; client modules only have to call this
macro before accessing the C API.

The exporting module is a modification of the spam module from
section A Simple Example. The function spam.system() does not
call the C library function system() directly, but a function
PySpam_System(), which would of course do something more
complicated in reality (such as adding “spam” to every command).
This function PySpam_System() is also exported to other extension
modules.

The function PySpam_System() is a plain C function, declared

```c
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function spam_system() is modified in a trivial way:

```c
static PyObject *
```
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}

In the beginning of the module, right after the line
#include "Python.h"

two more lines must be added:

#define SPAM_MODULE
#include "spammodule.h"

The #define is used to tell the header file that it is being included in
the exporting module, not a client module. Finally, the module's
initialization function must take care of initializing the C API pointer
array:

PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;
/* Initialize the C API pointer array */
PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

/* Create a Capsule containing the API pointer array */
c_api_object = PyCapsule_New((void *)PySpam_API, NULL);

if (c_api_object != NULL)
    PyModule_AddObject(m, "_C_API", c_api_object);
return m;

Note that PySpam_API is declared static; otherwise the pointer array would disappear when PyInit_spam() terminates!

The bulk of the work is in the header file spammodule.h, which looks like this:

#ifdef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1
All that a client module must do in order to have access to the function PySpam_System() is to call the function (or rather macro) import_spam() in its initialization function:
The main disadvantage of this approach is that the file spammodule.h is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section Capsules and in the implementation of Capsules (files Include/pycapsule.h and Objects/pycapsule.c in the Python source code distribution).

Footnotes

[1] An interface for this function already exists in the standard module os — it was chosen as a simple and straightforward example.

[2] The metaphor of “borrowing” a reference is not completely correct: the owner still has a copy of the reference.
Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

These guarantees don’t hold when you use the “old” style calling convention — this is still found in much existing code.
2. Defining Extension Types: Tutorial

Python allows the writer of a C extension module to define new types that can be manipulated from Python code, much like the built-in `str` and `list` types. The code for all extension types follows a pattern, but there are some details that you need to understand before you can get started. This document is a gentle introduction to the topic.
2.1. The Basics

The CPython runtime sees all Python objects as variables of type `PyObject*`, which serves as a “base type” for all Python objects. The `PyObject` structure itself only contains the object’s reference count and a pointer to the object’s “type object”. This is where the action is; the type object determines which (C) functions get called by the interpreter when, for instance, an attribute gets looked up on an object, a method called, or it is multiplied by another object. These C functions are called “type methods”.

So, if you want to define a new extension type, you need to create a new type object.

This sort of thing can only be explained by example, so here’s a minimal, but complete, module that defines a new type named Custom inside a C extension module `custom`:

```
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
}
```

**Note:** What we’re showing here is the traditional way of defining static extension types. It should be adequate for most uses. The C API also allows defining heap-allocated extension types using the `PyType_FromSpec()` function, which isn’t covered in this tutorial.
Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the previous chapter. This file defines three things:

1. What a Custom object contains: this is the `CustomObject` struct, which is allocated once for each Custom instance.
2. How the Custom **type** behaves: this is the `CustomType` struct, which defines a set of flags and function pointers that the interpreter inspects when specific operations are requested.

3. How to initialize the custom module: this is the `PyInit_custom` function and the associated `custommodule` struct.

The first bit is:

```c
typedef struct {
    PyObject_HEAD
} CustomObject;
```

This is what a Custom object will contain. `PyObject_HEAD` is mandatory at the start of each object struct and defines a field called `ob_base` of type `PyObject`, containing a pointer to a type object and a reference count (these can be accessed using the macros `Py_REFCNT` and `Py_TYPE` respectively). The reason for the macro is to abstract away the layout and to enable additional fields in debug builds.

**Note:** There is no semicolon above after the `PyObject_HEAD` macro. Be wary of adding one by accident: some compilers will complain.

Of course, objects generally store additional data besides the standard `PyObject_HEAD` boilerplate; for example, here is the definition for standard Python floats:

```c
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```
The second bit is the definition of the type object.

```c
static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_new = PyType_GenericNew,
};
```

**Note:** We recommend using C99-style designated initializers as above, to avoid listing all the `PyTypeObject` fields that you don’t care about and also to avoid caring about the fields’ declaration order.

The actual definition of `PyTypeObject` in `object.h` has many more fields than the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

We’re going to pick it apart, one field at a time:

```c
PyVarObject_HEAD_INIT(NULL, 0)
```

This line is mandatory boilerplate to initialize the `ob_base` field mentioned above.

```c
.tp_name = "custom.Custom",
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:
Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is custom and the type is Custom, so we set the type name to custom.Custom. Using the real dotted import path is important to make your type compatible with the pydoc and pickle modules.

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Cus
```

This is so that Python knows how much memory to allocate when creating new Custom instances. `tp_itemsize` is only used for variable-sized objects and should otherwise be zero.

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

Note: If you want your type to be subclassable from Python, and your type has the same `tp_basicsize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type's `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_basicsize` than its base type does. Most of the time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

We set the class flags to `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```
All types should include this constant in their flags. It enables all of the members defined until at least Python 3.3. If you need further members, you will need to OR the corresponding flags.

We provide a doc string for the type in `tp_doc`.

```
.tp_doc = "Custom objects",
```

To enable object creation, we have to provide a `tp_new` handler. This is the equivalent of the Python method `__new__()`, but has to be specified explicitly. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

Everything else in the file should be familiar, except for some code in `PyInit_custom()`:

```
if (PyType_Ready(&CustomType) < 0)
    return;
```

This initializes the Custom type, filling in a number of members to the appropriate default values, including `ob_type` that we initially set to `NULL`.

```
PyModule_AddObject(m, "Custom", (PyObject *) &CustomType);
```

This adds the type to the module dictionary. This allows us to create Custom instances by calling the Custom class:
>>> import custom
>>> mycustom = custom.Custom()

That's it! All that remains is to build it; put the above code in a file called custom.c and:

```python
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
ext_modules=[Extension("custom", ["custom.c"]))
```

in a file called setup.py; then typing

```
$ python setup.py build
```

at a shell should produce a file custom.so in a subdirectory; move to that directory and fire up Python — you should be able to import custom and play around with Custom objects.

That wasn't so hard, was it?

Of course, the current Custom type is pretty uninteresting. It has no data and doesn’t do anything. It can't even be subclassed.

**Note:** While this documentation showcases the standard distutils module for building C extensions, it is recommended in real-world use cases to use the newer and better-maintained setuptools library. Documentation on how to do this is out of scope for this document and can be found in the Python Packaging User’s Guide.
2.2. Adding data and methods to the Basic example

Let's extend the basic example to add some data and methods. Let's also make the type usable as a base class. We'll create a new module, custom2 that adds these capabilities:

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *istate)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
        }
    }
    return (PyObject *) self;
}
```
return NULL;
}
self->last = PyUnicode_FromString("");  
if (self->last == NULL) {
    Py_DECREF(self);
    return NULL;
}
    self->number = 0;
}
return (PyObject *) self;

static int
CustomInit(CustomObject *self, PyObject *args, PyObject *kwds) {
    static char *kwlist[] = {"first", "last", "number"};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi",
                                       &first, &last,
                                       &self->number))
    
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XINCREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first)
        "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last)
        "last name"},
    {"number", T_INT, offsetof(CustomObject, number)
        "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(
{  
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first
            return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last"
            return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
        "Return the name, combining the first and last
    },
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
This version of the module has a number of changes.

We’ve added an extra include:
This include provides declarations that we use to handle attributes, as described a bit later.

The Custom type now has three data attributes in its C struct, `first`, `last`, and `number`. The `first` and `last` variables are Python strings containing first and last names. The `number` attribute is a C integer.

The object structure is updated accordingly:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```
static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

which is assigned to the `tp_dealloc` member:

```
.tp_dealloc = (destructor) Custom_dealloc,
```

This method first clears the reference counts of the two Python
attributes. \texttt{Py\_XDECREF()} correctly handles the case where its argument is \texttt{NULL} (which might happen here if \texttt{tp\_new} failed midway). It then calls the \texttt{tp\_free} member of the object's type (computed by \texttt{Py\_TYPE(self)}) to free the object's memory. Note that the object's type might not be \texttt{CustomType}, because the object may be an instance of a subclass.

\begin{quote}
\textbf{Note:} The explicit cast to destructor above is needed because we defined \texttt{Custom\_dealloc} to take a \texttt{CustomObject *} argument, but the \texttt{tp\_dealloc} function pointer expects to receive a \texttt{PyObject *} argument. Otherwise, the compiler will emit a warning. This is object-oriented polymorphism, in C!
\end{quote}

We want to make sure that the first and last names are initialized to empty strings, so we provide a \texttt{tp\_new} implementation:

\begin{verbatim}
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *self)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString(""");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString(""");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
}
\end{verbatim}
and install it in the `tp_new` member:

```
.tp_new = Custom_new,
```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the `Custom` type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-`NULL` default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

**Note:** `tp_new` shouldn’t call `tp_init` explicitly, as the interpreter will do it itself.

The `tp_new` implementation calls the `tp_alloc` slot to allocate memory:

```
self = (CustomObject *) type->tp_alloc(type, 0);
```
Since memory allocation may fail, we must check the `tp_alloc` result against `NULL` before proceeding.

**Note:** We didn’t fill the `tp_alloc` slot ourselves. Rather, `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

**Note:** If you are creating a co-operative `tp_new` (one that calls a base type’s `tp_new` or `__new__()`) you must *not* try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We also define an initialization function which accepts arguments to provide initial values for our instance:

```c
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *{
    static char *kwlist[] = {"first", "last", "number"};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", &first, &last, &self->number))
        return -1;

    if (first) {
```
tmp = self->first;
Py_INCREF(first);
self->first = first;
Py_XDECREF(tmp);
}
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

by filling the `tp_init` slot.

```python
.tp_init = (initproc) Custom_init,
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Initializers always accept positional and keyword arguments, and they should return either 0 on success or -1 on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the `pickle` module by default doesn't call `__init__()` on unpickled instances). It can also be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this:

```python
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
```
self->first = first;
}

But this would be risky. Our type doesn’t restrict the type of the first member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the first member; or that destructor could release the Global interpreter Lock and let arbitrary code run in other threads that accesses and modifies our object.

To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don’t we have to do this?

- when we absolutely know that the reference count is greater than 1;
- when we know that deallocation of the object [1] will neither release the GIL nor cause any calls back into our type’s code;
- when decrementing a reference count in a `tp_dealloc` handler on a type which doesn’t support cyclic garbage collection [2].

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions:

```c
static PyMemberDef Custom_members[] = {
  "first", T_OBJECT_EX, offsetof(CustomObject, first), "first name"},
  "last", T_OBJECT_EX, offsetof(CustomObject, last), "last name"},
  "number", T_INT, offsetof(CustomObject, number), "custom number"},
  {NULL} /* Sentinel */
};
```
and put the definitions in the `tp_members` slot:

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the **Generic Attribute Management** section below for details.

A disadvantage of this approach is that it doesn’t provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to `NULL`. Even though we can make sure the members are initialized to non-`NULL` values, the members can be set to `NULL` if the attributes are deleted.

We define a single method, `Custom.name()`, that outputs the objects name as the concatenation of the first and last names.

```
static PyObject *
Custom_name(CustomObject *self)
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first"
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last"
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first
```

The method is implemented as a C function that takes a Custom (or
Custom subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

```python
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our first and last members are NULL. This is because they can be deleted, in which case they are set to NULL. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Now that we've defined the method, we need to create an array of method definitions:

```c
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS, 
     "Return the name, combining the first and last
    },
    {NULL} /* Sentinel */
};
```

(note that we used the METH_NOARGS flag to indicate that the method is expecting no arguments other than self)

and assign it to the tp_methods slot:

```c
.tp_methods = Custom_methods,
```

Finally, we'll make our type usable as a base class for subclassing.
We’ve written our methods carefully so far so that they don’t make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE
```

We rename `PyInit_custom()` to `PyInit_custom2()`, update the module name in the `PyModuleDef` struct, and update the full class name in the `PyTypeObject` struct.

Finally, we update our `setup.py` file to build the new module:

```python
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
          Extension("custom", ["custom.c"]),
          Extension("custom2", ["custom2.c"]),
      ])
```
2.3. Providing finer control over data attributes

In this section, we'll provide finer control over how the first and last attributes are set in the Custom example. In the previous version of our module, the instance variables first and last could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *keywds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString(""");
    }
```
if (self->first == NULL) {
    Py_DECREF(self);
    return NULL;
}
self->last = PyUnicode_FromString("");
if (self->last == NULL) {
    Py_DECREF(self);
    return NULL;
}
self->number = 0;
return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *keywds) {
    static char *kwlist[] = {"first", "last", "number"};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|Ui",
                                      &first, &last, &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
}
return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure) {
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, Py_ssize_t size) {
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value,
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The last attribute value must be a string");
        return -1;
    }
    tmp = self->last;
    Py_INCREF(value);
    self->last = value;
    Py_DECREF(tmp);
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst, "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast, "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED)
{
return PyUnicode_FromFormat("%S %S", self->first
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
        "Return the name, combining the first and last
    },
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
To provide greater control, over the first and last attributes, we'll use custom getter and setter functions. Here are the functions for getting and setting the first attribute:

```c
static PyObject *
Custom_getfirst(CustomObject *self, void *closure) {
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure) {
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                         "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    return 0;
}
```
Py_INCREF(value);
self->first = value;
Py_DECREF(tmp);
return 0;
}

The getter function is passed a Custom object and a “closure”, which is a void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the Custom object, the new value, and the closure. The new value may be NULL, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if its new value is not a string.

We create an array of PyGetSetDef structures:

```c
static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst, "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast, "last name", NULL},
    {NULL} /* Sentinel */
};
```

and register it in the tp_getset slot:

```c
.tp_getset = Custom_getsetters,
```

The last item in a PyGetSetDef structure is the “closure” mentioned
above. In this case, we aren’t using a closure, so we just pass `NULL`.

We also remove the member definitions for these attributes:

```c
static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number),
     "custom number"},
    {NULL} /* Sentinel */
};
```

We also need to update the `tp_init` handler to only allow strings [3] to be passed:

```c
static int Custom_init(CustomObject *self, PyObject *args, PyObject *
{
    static char *kwlist[] = {"first", "last", "number"};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi",
    &first, &last,
    &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
```
With these changes, we can assure that the first and last members are never NULL so we can remove checks for NULL values in almost all cases. This means that most of the `Py_DECREF()` calls can be converted to `Py_XDECREF()` calls. The only place we can’t change these calls is in the `tp_dealloc` implementation, where there is the possibility that the initialization of these members failed in `tp_new`.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the `setup.py` file.
2.4. Supporting cyclic garbage collection

Python has a cyclic garbage collector (GC) that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```python
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the Custom example, we allowed any kind of object to be stored in the first or last attributes [4]. Besides, in the second and third versions, we allowed subclassing Custom, and subclasses may add arbitrary attributes. For any of those two reasons, Custom objects can participate in cycles:

```python
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

To allow a Custom instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our Custom type needs to fill two additional slots and to enable a flag that enables these slots:


```c
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(CustomObject *self, visitproc visit,
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *
{
CustomObject *self;
self = (CustomObject *) type->tp_alloc(type, 0);
if (self != NULL) {
    self->first = PyUnicode_FromString("");
    if (self->first == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->last = PyUnicode_FromString("");
    if (self->last == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->number = 0;
}
return (PyObject *) self;

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *
{  
    static char *kwlist[] = {"first", "last", "number"};
    PyObject *first = NULL, *last = NULL, *tmp;
    if (!PyArg_ParseTupleAndKeywords(args, kwds, |UUi|
        &first, &last,
        &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
}
Py_INCREF(last);
    self->last = last;
    Py_DECREF(tmp);
}
return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number),
        "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value,
{  
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
    return 0;
}
static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value,
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError, "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->last);
    self->last = value;
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst, "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast, "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(0))
{
return PyUnicode_FromFormat("%S %S", self->first
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles:

```c
static int Custom_traverse(CustomObject *self, visitproc visit, {
   int vret;
   if (self->first) {
      vret = visit(self->first, arg);
      if (vret != 0)
         return vret;
   }
   if (self->last) {
      vret = visit(self->last, arg);
      if (vret != 0)
         return vret;
   }
   return 0;
}
```

For each subobject that can participate in cycles, we need to call the visit() function, which is passed to the traversal method. The
visit() function takes as arguments the subobject and the extra argument arg passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python provides a Py_VISIT() macro that automates calling visit functions. With Py_VISIT(), we can minimize the amount of boilerplate in Custom_traverse:

```c
static int
Custom_traverse(CustomObject *self, visitproc visit, {
    Py_VISIT(self->first);  
    Py_VISIT(self->last);  
    return 0;  
}
```

**Note:** The tp_traverse implementation must name its arguments exactly visit and arg in order to use Py_VISIT().

Second, we need to provide a method for clearing any subobjects that can participate in cycles:

```c
static int
Custom_clear(CustomObject *self)  
{
    Py_CLEAR(self->first);  
    Py_CLEAR(self->last);  
    return 0;  
}
```

Notice the use of the Py_CLEAR() macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call Py_XDECREF() instead on
the attribute before setting it to \texttt{NULL}, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (especially if there is a reference cycle).

\textbf{Note:} You could emulate \texttt{Py_CLEAR()} by writing:

\begin{verbatim}
PyObject *tmp;
tmp = self->first;
sel->first = NULL;
Py_XDECREF(tmp);
\end{verbatim}

Nevertheless, it is much easier and less error-prone to always use \texttt{Py_CLEAR()} when deleting an attribute. Don’t try to micro-optimize at the expense of robustness!

The deallocator \texttt{Custom_dealloc} may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling \texttt{PyObject_GC_UnTrack()} before clearing members. Here is our reimplemented deallocator using \texttt{PyObject_GC_UnTrack()} and \texttt{Custom_clear}:

\begin{verbatim}
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
\end{verbatim}

Finally, we add the \texttt{Py_TPFLAGS_HAVE_GC} flag to the class flags:
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE

That's pretty much it. If we had written custom `tp_alloc` or `tp_free` handlers, we'd need to modify them for cyclic garbage collection. Most extensions will use the versions automatically provided.
2.5. Subclassing other types

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built in types, since an extension can easily use the PyTypeObject it needs. It can be difficult to share these PyTypeObject structures between extension modules.

In this example we will create a SubList type that inherits from the built-in list type. The new type will be completely compatible with regular lists, but will have an additional increment() method that increases an internal counter:

```python
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

```c
#include <Python.h>

typedef struct {
    PyListObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
}
```
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", (PyCFunction) SubList_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int SubList_init(SubListObject *self, PyObject *args, PyObject *)
{
    if (PyList_Type.tp_init((PyObject *) self, args, Py)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist/SubList",
    .tp_doc = "SubList objects",
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASE |
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type."
    .m_size = -1,
};

PyMODINIT_FUNC
As you can see, the source code closely resembles the Custom examples in previous sections. We will break down the main differences between them.

The primary difference for derived type objects is that the base type's object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a SubList instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `SubListObject *`:

```
static int
```
We see above how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type's concrete base class. Due to cross-platform compiler issues, you can't fill that field directly with a reference to `PyList_Type`; it should be done later in the module initialization function:

```python
PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
```
Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving an existing type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` – the allocation function from the base type will be inherited.

After that, calling `PyType_Ready()` and adding the type object to the module is the same as with the basic Custom examples.

Footnotes

[1] This is true when we know that the object is a basic type, like a string or a float.

[2] We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

[3] We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won’t call back into our objects, we can’t guarantee that deallocating an instance of a string subclass won’t call back into our objects.

[4] Also, even with our attributes restricted to strings instances, the user could pass arbitrary `str` subclasses and therefore still create reference cycles.
Extending and Embedding the Python Interpreter
3. Defining Extension Types: Assorted Topics

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format <module>.<name> */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */
    destructor tp_dealloc;
    printfunc tp_print;
    getattribfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */
    hashfunc tp_hash;
    ternaryfunc tp_call;
} PyTypeObject;
```
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
unsigned long tp_flags;

const char *tp_doc; /* Documentation string */

/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
Py_ssize_t tp_weaklistoffset;

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
 descrgetfunc tp_descr_get;
desctrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
Now that's a lot of methods. Don't worry too much though – if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields. It's often easiest to find an example that includes the fields you need and then change the values to suit your new type.

```c
const char *tp_name; /* For printing */
```

The name of the type – as mentioned in the previous chapter, this will appear in various places, almost entirely for diagnostic purposes. Try
to choose something that will be helpful in such a situation!

```c
Py_ssize_t tp_basicsize, tp_itemsize; /* For allocate */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, tuples) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```c
const char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods – the ones most extension types will implement.
3.1. Finalization and De-allocation

destructor tp_dealloc;

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, you can put it here. The object itself needs to be freed here as well. Here is an example of this function:

```c
static void
c newdatatype_dealloc(newdatatypeobject *obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```c
static void
my_dealloc(PyObject *obj)
```
```c
MyObject *self = (MyObject *) obj;
PyObject *cbresult;

if (self->my_callback != NULL) {
    PyObject *err_type, *err_value, *err_traceback;

    /* This saves the current exception state */
    PyErr_Fetch(&err_type, &err_value, &err_traceback);

    cbresult = PyObject_CallObject(self->my_callback, NULL);
    if (cbresult == NULL)
        PyErr_WriteUnraisable(self->my_callback);
    else
        Py_DECREF(cbresult);

    /* This restores the saved exception state */
    PyErr_Restore(err_type, err_value, err_traceback);
    Py_DECREF(self->my_callback);
}
Py_TYPE(obj)->tp_free((PyObject*)self);
```

**Note:** There are limitations to what you can safely do in a deallocator function. First, if your type supports garbage collection (using `tp_traverse` and/or `tp_clear`), some of the object's members can have been cleared or finalized by the time `tp_dealloc` is called. Second, in `tp_dealloc`, your object is in an unstable state: its reference count is equal to zero. Any call to a non-trivial object or API (as in the example above) might end up calling `tp_dealloc` again, causing a double free and a crash.

Starting with Python 3.4, it is recommended not to put any complex
finalization code in `tp_dealloc`, and instead use the new `tp_finalize` type method.

**See also:** PEP 442 explains the new finalization scheme.
3.2. Object Presentation

In Python, there are two ways to generate a textual representation of an object: the `repr()` function, and the `str()` function. (The `print()` function just calls `str()`. ) These handlers are both optional.

```c
reprfunc tp_repr;
reprfunc tp_str;
```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

```c
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Repr-ified_newdatatype\nobj->obj_UnderlyingDatatypePtr");
}
```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely-identifying value for the object.

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.
Here is a simple example:

```c
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Stringified_newdatatype{%d}%d", obj->obj_UnderlyingDatatypePtr);
}
```
3.3. Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is *NULL*.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a char*, while the other accepts a *PyObject*. Each type can use whichever pair makes more sense for the implementation's convenience.

```c
getattrfunc tp_getattr;  /* char * version */
setattrfunc tp_setattr;  /* ... */
getattrofunc tp_getattro; /* PyObject * version */
setattrofunc tp_setattro;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the *PyObject* version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

3.3.1. Generic Attribute Management
Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when `PyType_Ready()` is called.
2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptors* which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields `NULL` as well, allowing the base type to handle attributes.

The tables are declared as three fields of the type object:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not `NULL`, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure:

```
typedef struct PyMethodDef {
```
One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The ml_name field of the sentinel must be `NULL`.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```c
typedef struct PyMemberDef {
    const char *name;
    int type;
    int offset;
    int flags;
    const char *doc;
} PyMemberDef;
```

For each entry in the table, a descriptor will be constructed and added to the type which will be able to extract a value from the instance structure. The type field should contain one of the type codes defined in the structmember.h header; the value will be used to determine how to convert Python values to and from C values. The flags field is used to store flags which control how the attribute can be accessed.

The following flag constants are defined in structmember.h; they may be combined using bitwise-OR.
<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READONLY</td>
<td>Never writable.</td>
</tr>
<tr>
<td>READ_RESTRICTED</td>
<td>Not readable in restricted mode.</td>
</tr>
<tr>
<td>WRITE_RESTRICTED</td>
<td>Not writable in restricted mode.</td>
</tr>
<tr>
<td>RESTRICTED</td>
<td>Not readable or writable in restricted mode.</td>
</tr>
</tbody>
</table>

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

As with the `tp_methods` table, a sentinel entry with a name value of `NULL` is required.

### 3.3.2. Type-specific Attribute Management

For simplicity, only the `char*` version will be demonstrated here; the type of the name parameter is the only difference between the `char*` and `PyObject*` flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. It explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__() method of a class would be called.
Here is an example:

```c
static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    if (strcmp(name, "data") == 0)
    {
        return PyLong_FromLong((long)obj->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%.400s'",
                 tp->tp_name, name);

    return NULL;
}
```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```c
static int
newdatatype_setattr(newdatatypeobject *obj, char *name)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute:	%s"
                 return -1;
}
```

3.4. Object Comparison

```python
richcmpfunc tp_richcompare;
```

The `tp_richcompare` handler is called when comparisons are needed. It is analogous to the rich comparison methods, like `__lt__()`, and also called by `PyObject_RichCompare()` and `PyObject_RichCompareBool()`.

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GT`, `Py_LT` or `Py_GT`. It should compare the two objects with respect to the specified operator and return `Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object’s comparison method should be tried, or `NULL` if an exception was set.

Here is a sample implementation, for a datatype that is considered equal if the size of an internal pointer is equal:

```python
static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2,
{
    PyObject *result;
    int c, size1, size2;

    /* code to make sure that both arguments are of newdatatype omitted */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
```
case Py_LT:  c = size1 < size2; break;
case Py_LE:  c = size1 <= size2; break;
case Py_EQ:  c = size1 == size2; break;
case Py_NE:  c = size1 != size2; break;
case Py_GT:  c = size1 > size2; break;
case Py_GE:  c = size1 >= size2; break;
}
result = c ? Py_True : Py_False;
Py_INCREF(result);
return result;
3.5. Abstract Protocol Support

Python supports a variety of *abstract* ‘protocols;’ the specific interfaces provided to use these interfaces are documented in *Abstract Objects Layer*.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```c
PyNumberMethods   *tp_as_number;
PySequenceMethods  *tp_as_sequence;
PyMappingMethods   *tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type *PyNumberMethods*, *PySequenceMethods*, or *PyMappingMethods*, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the *Objects* directory of the Python source distribution.

```python
hashfunc tp_hash;
```
This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a simple example:

```c
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
    return result;
}
```

Py_hash_t is a signed integer type with a platform-varying width. Returning -1 from tp_hash indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```c
ternaryfunc tp_call;
```

This function is called when an instance of your data type is “called”, for example, if obj1 is an instance of your data type and the Python script contains obj1('hello'), the tp_call handler is invoked.

This function takes three arguments:

1. `self` is the instance of the data type which is the subject of the call. If the call is obj1('hello'), then `self` is obj1.
2. `args` is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. `kwds` is a dictionary of keyword arguments that were passed. If this is non-NULL and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments.
If you do not want to support keyword arguments and this is non-
NULL, raise a **TypeError** with a message saying that keyword
arguments are not supported.

Here is a toy *tp_call* implementation:

```c
static PyObject *
newdatatype_call(newdatatypeobject *self, PyObject *
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &
        return NULL;
    } result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3:
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
        return result;
)

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Both handlers
take exactly one parameter, the instance for which they are being
called, and return a new reference. In the case of an error, they should
set an exception and return **NULL**. *tp_iter* corresponds to the
Python **__iter__()** method, while *tp_iternext* corresponds to
the Python **__next__()** method.
Any iterable object must implement the `tp_iter` handler, which must return an iterator object. Here the same guidelines apply as for Python classes:

- For collections (such as lists and tuples) which can support multiple independent iterators, a new iterator should be created and returned by each call to `tp_iter`.
- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement `tp_iter` by returning a new reference to themselves – and should also therefore implement the `tp_iternext` handler.

Any iterator object should implement both `tp_iter` and `tp_iternext`. An iterator’s `tp_iter` handler should return a new reference to the iterator. Its `tp_iternext` handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, `tp_iternext` may return `NULL` without setting an exception, or it may set `StopIteration` in addition to returning `NULL`; avoiding the exception can yield slightly better performance. If an actual error occurs, `tp_iternext` should always set an exception and return `NULL`. 
3.6. Weak Reference Support

One of the goals of Python’s weak reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on performance-critical objects (such as numbers).

See also: Documentation for the weakref module.

For an object to be weakly referencable, the extension type must do two things:

1. Include a PyObject* field in the C object structure dedicated to the weak reference mechanism. The object’s constructor should leave it NULL (which is automatic when using the default tp_alloc).
2. Set the tp_weaklistoffset type member to the offset of the aforementioned field in the C object structure, so that the interpreter knows how to access and modify that field.

Concretely, here is how a trivial object structure would be augmented with the required field:

```c
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist; /* List of weak references */
} TrivialObject;
```

And the corresponding member in the statically-declared type object:

```c
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
};
```
The only further addition is that `tp_dealloc` needs to clear any weak references (by calling `PyObject_ClearWeakRefs()`) if the field is non-NULL:

```c
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```
3.7. More Suggestions

In order to learn how to implement any specific method for your new data type, get the CPython source code. Go to the `objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_richcompare`). You will find examples of the function you want to implement.

When you need to verify that an object is a concrete instance of the type you are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following:

```c
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything";
    return NULL;
}
```

See also:

Download CPython source releases.
  https://www.python.org/downloads/source/

The CPython project on GitHub, where the CPython source code is developed.
  https://github.com/python/cpython
4. Building C and C++ Extensions

A C extension for CPython is a shared library (e.g. a .so file on Linux, .pyd on Windows), which exports an *initialization function*.

To be importable, the shared library must be available on `PYTHONPATH`, and must be named after the module name, with an appropriate extension. When using distutils, the correct filename is generated automatically.

The initialization function has the signature:

```
PyObject* PyInit_modulename(void)
```

It returns either a fully-initialized module, or a `PyModuleDef` instance. See *Initializing C modules* for details.

For modules with ASCII-only names, the function must be named `PyInit_<modulename>`, with `<modulename>` replaced by the name of the module. When using *Multi-phase initialization*, non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<modulename>`, with `<modulename>` encoded using Python’s *punycode* encoding with hyphens replaced by underscores. In Python:

```python
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_').replace(b'_', b'.').replace(b'.', b'_')
    return b'PyInit' + suffix
```
It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the “Multiple modules in one library” section in PEP 489 for details.
4.1. Building C and C++ Extensions with distutils

Extension modules can be built using distutils, which is included in Python. Since distutils also supports creation of binary packages, users don't necessarily need a compiler and distutils to install the extension.

A distutils package contains a driver script, `setup.py`. This is a plain Python file, which, in the most simple case, could look like this:

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
                     sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

With this `setup.py`, and a file `demo.c`, running

```
python setup.py build
```

will compile `demo.c`, and produce an extension module named `demo` in the `build` directory. Depending on the system, the module file will end up in a subdirectory `build/lib.system`, and may have a name like `demo.so` or `demo.pyd`.

In the `setup.py`, all execution is performed by calling the `setup` function. This takes a variable number of keyword arguments, of which
the example above uses only a subset. Specifically, the example specifies meta-information to build packages, and it specifies the contents of the package. Normally, a package will contain additional modules, like Python source modules, documentation, subpackages, etc. Please refer to the distutils documentation in "Distributing Python Modules (Legacy version)" to learn more about the features of distutils; this section explains building extension modules only.

It is common to pre-compute arguments to setup(), to better structure the driver script. In the example above, the ext_modules argument to setup() is a list of extension modules, each of which is an instance of the Extension. In the example, the instance defines an extension named demo which is build by compiling a single source file, demo.c.

In many cases, building an extension is more complex, since additional preprocessor defines and libraries may be needed. This is demonstrated in the example below.

```python
from distutils.core import setup, Extension

module1 = Extension('demo',
    define_macros = [(
        'MAJOR_VERSION', 1),
        ('MINOR_VERSION', 0)
    ],
    include_dirs = ['/usr/local/incl'],
    libraries = ['tcl83'],
    library_dirs = ['/usr/local/lib'],
    sources = ['demo.c'])

setup (name = 'PackageName',
    version = '1.0',
    description = 'This is a demo package',
    author = 'Martin v. Loewis',
    author_email = 'martin@v.loewis.de',
)```
url = 'https://docs.python.org/extending/building'
long_description = '''
This is really just a demo package.
''',
ext_modules = [module1])

In this example, `setup()` is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, distutils passes this information in different ways to the compiler. For example, on Unix, this may result in the compilation commands

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC
gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/
```

These lines are for demonstration purposes only; distutils users should trust that distutils gets the invocations right.
4.2. Distributing your extension modules

When an extension has been successfully build, there are three ways to use it.

End-users will typically want to install the module, they do so by running

```
python setup.py install
```

Module maintainers should produce source packages; to do so, they run

```
python setup.py sdist
```

In some cases, additional files need to be included in a source distribution; this is done through a MANIFEST.in file; see [Specifying the files to distribute](https://docs.python.org/3.7/extending-and-embedding/index.html#specifying-the-files-to-distribute) for details.

If the source distribution has been build successfully, maintainers can also create binary distributions. Depending on the platform, one of the following commands can be used to do so.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```
5. Building C and C++ Extensions on Windows

This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the Unix programmer interested in producing software which can be successfully built on both Unix and Windows.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

**Note:** This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as XY; in practice, 'X' will be the major version number and 'Y' will be the minor version number of the Python release you’re working with. For example, if you are using Python 2.2.1, XY will actually be 22.
5.1. A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on Unix: use the distutils package to control the build process, or do things manually. The distutils approach works well for most extensions; documentation on using distutils to build and package extension modules is available in Distributing Python Modules (Legacy version). If you find you really need to do things manually, it may be instructive to study the project file for the winsound standard library module.
5.2. Differences Between Unix and Windows

Unix and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In Unix, a shared object (.so) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file’s code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (.dll) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program’s memory; instead, the code already uses the DLL’s lookup table, and the lookup table is modified at runtime to point to the functions and data.

In Unix, there is only one type of library file (.a) which contains code from several object files (.o). During the link step to create a shared object file (.so), the linker may find that it doesn’t know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called .lib). A static library is like a Unix .a file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is
legal, and will be present in the program when the DLL is loaded. So
the linker uses the information from the import library to build the
lookup table for using identifiers that are not included in the DLL.
When an application or a DLL is linked, an import library may be
generated, which will need to be used for all future DLLs that depend
on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which
should share another block of code A. On Unix, you would not pass
A.a to the linker for B.so and C.so; that would cause it to be
included twice, so that B and C would each have their own copy. In
Windows, building A.dll will also build A.lib. You do pass A.lib
to the linker for B and C. A.lib does not contain code; it just contains
information which will be used at runtime to access A's code.

In Windows, using an import library is sort of like using import spam;
it gives you access to spam's names, but does not create a separate
copy. On Unix, linking with a library is more like from spam import *
; it does create a separate copy.
5.3. Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to). The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you must pass pythonXY.lib to the linker. To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

The first command created three files: spam.obj, spam.dll and spam.lib. Spam.dll does not contain any Python functions (such as `PyArg_ParseTuple()`), but it does know how to find the Python code thanks to pythonXY.lib.

The second command created ni.dll (and .obj and .lib), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say `_declspec(dllexport)`, as in `void _declspec(dllexport) initspam(void)` or `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify `ignore default`
*libraries*. Add the correct `msvcrtxx.lib` to the list of libraries.
1. Embedding Python in Another Application

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python — instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()`. There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to `PyRun_SimpleString()`, or
you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

See also:

**Python/C API Reference Manual**

The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.
1.1. Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```c
#include <Python.h>

int main(int argc, char *argv[]) {
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]");
        exit(1);
    }
    Py_SetProgramName(program); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime
    print('Today is', ctime(time()))
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

The `Py_SetProgramName()` function should be called before `Py_Initialize()` to inform the interpreter about paths to Python run-time libraries. Next, the Python interpreter is initialized with `Py_Initialize()`, followed by the execution of a hard-coded Python script that prints the date and time. Afterwards, the
`Py_FinalizeEx()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.
1.2. Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and
vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.
1.3. Pure Embedding

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

```c
#include <Python.h>

int main(int argc, char *argv[]) {
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]
            return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            ...
pArgs = PyTuple_New(argc - 3);
for (i = 0; i < argc - 3; ++i) {
    pValue = PyLong_FromLong(atoi(argv[i]);
    if (!pValue) {
        Py_DECREF(pArgs);
        Py_DECREF(pModule);
        fprintf(stderr, "Cannot convert
    return 1;

    /* pValue reference stolen here: */
    PyTuple_SetItem(pArgs, i, pValue);
}
    pValue = PyObject_CallObject(pFunc, pArgs);
    Py_DECREF(pArgs);
    if (pValue != NULL) {
        printf("Result of call: %ld\n", PyLong_AsLong(pValue));
        Py_DECREF(pValue);
    } else {
        Py_DECREF(pFunc);
        Py_DECREF(pModule);
        PyErr_Print();
        fprintf(stderr,"Call failed\n");
        return 1;
    }
} else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function \"
}
Py_XDECREF(pFunc);
Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load "%s\n", a
return 1;
}
if (Py_FinalizeEx() < 0) {
    return 120;
}
return 0;

This code loads a Python script using \texttt{argv[1]} and calls the function named in \texttt{argv[2]}. Its integer arguments are the other values of the \texttt{argv} array. If you compile and link this program (let's call the finished executable \texttt{call}), and use it to execute a Python script, such as:

\begin{verbatim}
def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c
\end{verbatim}

then the result should be:

\begin{verbatim}
$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6
\end{verbatim}

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

\begin{verbatim}
Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
\end{verbatim}
pModule = PyImport_Import(pName);

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyUnicode_FromString()` data conversion routine.

```
pFunc = PyObject_GetAttrString(pModule, argv[2]); /* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we're looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.
1.4. Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```c
static int numargs=0;

/* Return the number of arguments of the application */
static PyObject *
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
        "Return the number of arguments received by the
    {NULL, NULL, 0, NULL}
};

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL, NULL
};
```
static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}

Insert the above code just above the main() function. Also, insert the following two statements before the call to Py_Initialize():

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```
import emb
print("Number of arguments", emb.numargs())
```

In a real application, the methods will expose an API of the application to Python.
1.5. Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++. 
1.6. Compiling and Linking under Unix-like systems

It is not necessarily trivial to find the right flags to pass to your compiler (and linker) in order to embed the Python interpreter into your application, particularly because Python needs to load library modules implemented as C dynamic extensions (.so files) linked against it.

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python3-config` script may also be available). This script has several options, of which the following will be directly useful to you:

- `pythonX.Y-config --cflags` will give you the recommended flags when compiling:

  ```
  $ /opt/bin/python3.4-config --cflags
  -I/opt/include/python3.4m -I/opt/include/python3
  ```

- `pythonX.Y-config --ldflags` will give you the recommended flags when linking:

  ```
  $ /opt/bin/python3.4-config --ldflags
  -L/opt/lib/python3.4/config-3.4m -lpthread -ldl
  ```

**Note:** To avoid confusion between several Python installations (and especially between the system Python and your own compiled Python), it is recommended that you use the absolute path to `pythonX.Y-config`, as in the above example.
If this procedure doesn’t work for you (it is not guaranteed to work for all Unix-like platforms; however, we welcome bug reports) you will have to read your system’s documentation about dynamic linking and/or examine Python’s Makefile (use sysconfig.get_makefile_filename() to find its location) and compilation options. In this case, the sysconfig module is a useful tool to programmatically extract the configuration values that you will want to combine together. For example:

```python
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```
Introduction

The Application Programmer’s Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write extension modules for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as embedding Python in an application.

Writing an extension module is a relatively well-understood process, where a “cookbook” approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you’re embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it’s probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.
Coding standards

If you’re writing C code for inclusion in CPython, you must follow the guidelines and standards defined in PEP 7. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.
Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```
#include "Python.h"
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

**Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include `Python.h` before any standard headers are included.

All user visible names defined by `Python.h` (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

**Important:** user code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's `configure` script and `version` is `%d.%d`
sys.version_info[:2]. On Windows, the headers are installed in `prefix/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do not place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `exec_prefix`.

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be `extern "C"`, so there is no need to do anything special to use the API from C++.
Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (e.g. `Py_RETURN_NONE`). Others of a more general utility are defined here. This is not necessarily a complete listing.

**Py_UNREACHABLE()**
Use this when you have a code path that you do not expect to be reached. For example, in the `default:` clause in a `switch` statement for which all possible values are covered in `case` statements. Use this in places where you might be tempted to put an `assert(0)` or `abort()` call.

*New in version 3.7.*

**Py_ABS(x)**
Return the absolute value of `x`.

*New in version 3.3.*

**Py_MIN(x, y)**
Return the minimum value between `x` and `y`.

*New in version 3.3.*

**Py_MAX(x, y)**
Return the maximum value between `x` and `y`.

*New in version 3.3.*

**Py_STRINGIFY(x)**
Convert x to a C string. E.g. Py_STRINGIFY(123) returns "123".

New in version 3.4.

**Py_MEMBER_SIZE**(type, member)
Return the size of a structure (type) member in bytes.

New in version 3.6.

**Py_CHARMASK**(c)
Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns c cast to an unsigned char.

**Py_GETENV**(s)
Like getenv(s), but returns NULL if -E was passed on the command line (i.e. if Py_IGNORE_ENVIRONMENT_FLAG is set).

**Py_UNUSED**(arg)
Use this for unused arguments in a function definition to silence compiler warnings, e.g. PyObject* func(PyObject *Py_UNUSED(ignored)).

New in version 3.4.
Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type `PyObject*`. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type `PyObject`, only pointer variables of type `PyObject*` can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static `PyTypeObject` objects.

All Python objects (even Python integers) have a type and a reference count. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in The standard type hierarchy). For each of the well-known types there is a macro to check whether an object is of that type; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by `a` is a Python list.

Reference Counts

The reference count is important because today's computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object's reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other
objects may be deallocated in turn, if this decrement makes their reference count become zero, and so on. (There's an obvious problem with objects that reference each other here; for now, the solution is “don’t do that.”)

Reference counts are always manipulated explicitly. The normal way is to use the macro `Py_INCREF()` to increment an object’s reference count by one, and `Py_DECREF()` to decrement it by one. The `Py_DECREF()` macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object’s type structure. The type-specific deallocator takes care of decrementing the reference counts for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming `sizeof(Py_ssize_t) >= sizeof(void*)`). Thus, the reference count increment is a simple operation.

It is not necessary to increment an object’s reference count for every local variable that contains a pointer to an object. In theory, the object’s reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn’t changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily. An important situation where this arises is in objects that are passed as
arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count and possible deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a `Py_DECREF()`, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with `PyObject_`, `PyNumber_`, `PySequence_` or `PyMapping_`). These operations always increment the reference count of the object they return. This leaves the caller with the responsibility to call `Py_DECREF()` when they are done with the result; this soon becomes second nature.

Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned: they are always shared). “Owning a reference” means being responsible for calling `Py_DECREF` on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually decref’ing it by calling `Py_DECREF()` or `Py_XDECREF()` when it’s no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the
caller is said to receive a new reference. When no ownership is transferred, the caller is said to borrow the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes in a reference to an object, there are two possibilities: the function steals a reference to the object, or it does not. Stealing a reference means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are PyList_SetItem() and PyTuple_SetItem(), which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple (1, 2, "three") could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```python
PyObject *t;

    t = PyTuple_New(3);
    PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
    PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
    PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Here, PyLong_FromLong() returns a new reference which is immediately stolen by PyTuple_SetItem(). When you want to keep using an object although the reference to it will be stolen, use Py_INCREF() to grab another reference before calling the reference-stealing function.

Incidentally, PyTuple_SetItem() is the only way to set tuple items;
PySequence_SetItem() and PyObject_SetItem() refuse to do this since tuples are an immutable data type. You should only use PyTuple_SetItem() for tuples that you are creating yourself.

Equivalent code for populating a list can be written using PyList_New() and PyList_SetItem().

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There’s a generic function, Py_BuildValue(), that can create most common objects from C values, directed by a format string. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```c
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use PyObject_SetItem() and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding reference counts is much saner, since you don't have to increment a reference count so you can give a reference away (“have it be stolen”). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```c
int set_all(PyObject *target, PyObject *item) {
    Py_ssize_t i, n;

    n = PyObject_Contains(target); 
    if (n < 0)
```
The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like `PyObject_GetItem()` and `PySequence_GetItem()`, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — the plumage (the type of the object passed as an argument to the function) doesn’t enter into it! Thus, if you extract an item from a list using `PyList_GetItem()`, you don’t own the reference — but if you obtain the same item from the same list using `PySequence_GetItem()` (which happens to take exactly the same arguments), you do own a reference to the returned object.
Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.

```c
long sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail
        if (!PyLong_Check(item)) continue; /* Skip non-integers
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}

long sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    ```
for (i = 0; i < n; i++) {
    item = PySequence_GetItem(sequence, i);
    if (item == NULL)
        return -1; /* Not a sequence, or other failure */
    if (PyLong_Check(item)) {
        value = PyLong_AsLong(item);
        Py_DECREF(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long */
            return -1;
        total += value;
    } else {
        Py_DECREF(item); /* Discard reference ownership */
    }
}
return total;

Types

There are few other data types that play a significant role in the Python/C API; most are simple C types such as int, long, double and char*. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.
Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller’s caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function’s documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either NULL or -1, depending on the function’s return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with PyErr_Occurred(). These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function PyErr_Occurred() can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and NULL otherwise. There are a number of functions to set the exception state: PyErr_SetString() is the most common (though not the most general) function to set the exception state, and PyErr_Clear() clears the exception state.

The full exception state consists of three objects (all of which can be
null): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python result of `sys.exc_info()`; however, they are not the same: the Python objects represent the last exception being handled by a Python `try ... except` statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter’s main loop, which takes care of transferring it to `sys.exc_info()` and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function `sys.exc_info()`, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should not set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn’t need to clean up any owned references when it
detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```python
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Here is the corresponding C code, in all its glory:

```c
int incr_item(PyObject *dict, PyObject *key)
{
    /* Include Py_XDECREF.h */
    PyObject *item = NULL, *const_one = NULL, *incremented_item
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }

    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;
```
incremented_item = PyNumber_Add(item, const_one)
if (incremented_item == NULL)
        goto error;

if (PyObject_SetItem(dict, key, incremented_item)
        goto error;
rv = 0; /* Success */
/* Continue with cleanup code */

error:
    /* Cleanup code, shared by success and failure paths */
    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}

This example represents an endorsed use of the goto statement in C!
It illustrates the use of PyErr_ExceptionMatches() and PyErr_Clear() to handle specific exceptions, and the use of Py_XDECREF() to dispose of owned references that may be NULL (note the 'X' in the name; Py_DECREF() would crash when confronted with a NULL reference). It is important that the variables used to hold owned references are initialized to NULL for this to work; likewise, the proposed return value is initialized to -1 (failure) and only set to success after the final call made is successful.
Embedding Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `builtins`, `__main__`, and `sys`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the “script argument list” (`sys.argv`). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgvEx(argc, argv, updatepath)` after the call to `Py_Initialize()`.

On most systems (in particular, on Unix and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/pythonX.Y` relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable PATH).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in
The user can override this behavior by setting the environment variable `PYTHONHOME`, or insert additional directories in front of the standard path by setting `PYTHONPATH`.

The embedding application can steer the search by calling `Py_SetProgramName(file)` before calling `Py_Initialize()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in `Modules/getpath.c`).

Sometimes, it is desirable to “uninitialize” Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_FinalizeEx()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_FinalizeEx()` does not free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.
Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

A full list of the various types of debugging builds is in the file Misc/SpecialBuilds.txt in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently-used builds will be described in the remainder of this section.

Compiling the interpreter with the Py_DEBUG macro defined produces what is generally meant by “a debug build” of Python. Py_DEBUG is enabled in the Unix build by adding --with-pydebug to the .configure command. It is also implied by the presence of the not-Python-specific _DEBUG macro. When Py_DEBUG is enabled in the Unix build, compiler optimization is disabled.

In addition to the reference count debugging described below, the following extra checks are performed:

- Extra checks are added to the object allocator.
- Extra checks are added to the parser and compiler.
- Downcasts from wide types to narrow types are checked for loss of information.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a test_c_api() method.
- Sanity checks of the input arguments are added to frame creation.
- The storage for ints is initialized with a known invalid pattern to
catch reference to uninitialized digits.

- Low-level tracing and extra exception checking are added to the runtime virtual machine.
- Extra checks are added to the memory arena implementation.
- Extra debugging is added to the thread module.

There may be additional checks not mentioned here.

Defining Py_TRACE_REFS enables reference tracing. When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every PyObject. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.) Implied by Py_DEBUG.

Please refer to Misc/SpecialBuilds.txt in the Python source distribution for more detailed information.
Stable Application Binary Interface

Traditionally, the C API of Python will change with every release. Most changes will be source-compatible, typically by only adding API, rather than changing existing API or removing API (although some interfaces do get removed after being deprecated first).

Unfortunately, the API compatibility does not extend to binary compatibility (the ABI). The reason is primarily the evolution of struct definitions, where addition of a new field, or changing the type of a field, might not break the API, but can break the ABI. As a consequence, extension modules need to be recompiled for every Python release (although an exception is possible on Unix when none of the affected interfaces are used). In addition, on Windows, extension modules link with a specific pythonXY.dll and need to be recompiled to link with a newer one.

Since Python 3.2, a subset of the API has been declared to guarantee a stable ABI. Extension modules wishing to use this API (called "limited API") need to define Py_LIMITED_API. A number of interpreter details then become hidden from the extension module; in return, a module is built that works on any 3.x version (x>=2) without recompilation.

In some cases, the stable ABI needs to be extended with new functions. Extension modules wishing to use these new APIs need to set Py_LIMITED_API to the PY_VERSION_Hex value (see API and ABI Versioning) of the minimum Python version they want to support (e.g. 0x03030000 for Python 3.3). Such modules will work on all subsequent Python releases, but fail to load (because of missing symbols) on the older releases.
As of Python 3.2, the set of functions available to the limited API is documented in PEP 384. In the C API documentation, API elements that are not part of the limited API are marked as “Not part of the limited API.”
The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are Py_eval_input, Py_file_input, and Py_single_input. These are described following the functions which accept them as parameters.

Note also that several of these functions take FILE* parameters. One particular issue which needs to be handled carefully is that the FILE structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that FILE* parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

int Py_Main(int argc, wchar_t **argv)

The main program for the standard interpreter. This is made available for programs which embed Python. The argc and argv parameters should be prepared exactly as those which are passed to a C program’s main() function (converted to wchar_t according to the user’s locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be 0 if the interpreter exits normally (i.e., without an exception), 1 if the interpreter exits due to an exception, or 2 if the parameter list does not represent a
valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return 1, but exit the process, as long as `Py_InspectFlag` is not set.

```c
int PyRun_AnyFile(FILE *fp, const char *filename)
```
This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving `closeit` set to 0 and `flags` set to `NULL`.

```c
int PyRun_AnyFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
```
This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the `closeit` argument set to 0.

```c
int PyRun_AnyFileEx(FILE *fp, const char *filename, int closeit)
```
This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the `flags` argument set to `NULL`.

```c
int PyRun_AnyFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
```
If `fp` refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. `filename` is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If `filename` is `NULL`, this function uses `"???"` as the filename.

```c
int PyRun_SimpleString(const char *command)
```
This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the `PyCompilerFlags*` argument set to `NULL`.  


int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags)

Executes the Python source code from command in the __main__ module according to the flags argument. If __main__ does not already exist, it is created. Returns 0 on success or -1 if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of flags, see below.

Note that if an otherwise unhandled SystemExit is raised, this function will not return -1, but exit the process, as long as Py_InspectFlag is not set.

int PyRun_SimpleFile(FILE *fp, const char *filename)

This is a simplified interface to PyRun_SimpleFileExFlags() below, leaving closeit set to 0 and flags set to NULL.

int PyRun_SimpleFileEx(FILE *fp, const char *filename, int closeit)

This is a simplified interface to PyRun_SimpleFileExFlags() below, leaving flags set to NULL.

int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)

Similar to PyRun_SimpleStringFlags(), but the Python source code is read from fp instead of an in-memory string. filename should be the name of the file, it is decoded from the filesystem encoding (sys.getfilesystemencoding()). If closeit is true, the file is closed before PyRun_SimpleFileExFlags returns.

int PyRun_InteractiveOne(FILE *fp, const char *filename)

This is a simplified interface to
```c
PyRun_InteractiveOneFlags() below, leaving flags set to NULL.

int PyRun_InteractiveOneFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Read and execute a single statement from a file associated with an interactive device according to the flags argument. The user will be prompted using sys.ps1 and sys.ps2. filename is decoded from the filesystem encoding (sys.getfilesystemencoding()).

Returns 0 when the input was executed successfully, -1 if there was an exception, or an error code from the errcode.h include file distributed as part of Python if there was a parse error. (Note that errcode.h is not included by Python.h, so must be included specifically if needed.)

int PyRun_InteractiveLoop(FILE *fp, const char *filename)

This is a simplified interface to PyRun_InteractiveLoopFlags() below, leaving flags set to NULL.

int PyRun_InteractiveLoopFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using sys.ps1 and sys.ps2. filename is decoded from the filesystem encoding (sys.getfilesystemencoding()). Returns 0 at EOF or a negative number upon failure.

int (*PyOS_InputHook)(void)

Can be set to point to a function with the prototype int func(void). The function will be called when Python's interpreter
prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the Modules/_tkinter.c in the Python source code.

```
char* (*PyOS_ReadlineFunctionPointer)(FILE *, FILE *, const char *)
```

Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string `prompt` if it's not `NULL`, and then read a line of input from the provided standard input file, returning the resulting string. For example, The `readline` module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or `NULL` if an error occurred.

*Changed in version 3.4:* The result must be allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, instead of being allocated by `PyMem_Malloc()` or `PyMem_Realloc()`.

```
struct _node* PyParser_SimpleParseString(const char *str, int start)
```

This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving `filename` set to `NULL` and `flags` set to `0`.

```
struct _node* PyParser_SimpleParseStringFlags(const char *str, int start, int flags)
```

This is a simplified interface to
PyParser_SimpleParseStringFlagsFilename() below, leaving *filename* set to NULL.

```c
struct _node*
PyParser_SimpleParseStringFlagsFilename(const char *str, const char *filename, int start, int flags)
```

Parse Python source code from *str* using the start token *start* according to the *flags* argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

```c
struct _node*
PyParser_SimpleParseFile(FILE *fp, const char *filename, int start)
```

This is a simplified interface to PyParser_SimpleParseFileFlags() below, leaving *flags* set to 0.

```c
struct _node*
PyParser_SimpleParseFileFlags(FILE *fp, const char *filename, int start, int flags)
```

Similar to PyParser_SimpleParseStringFlagsFilename(), but the Python source code is read from *fp* instead of an in-memory string.

```c
PyObject*
PyRun_String(const char *str, int start,
PyObject *globals, PyObject *locals)
```

*Return value: New reference.*

This is a simplified interface to PyRun_StringFlags() below, leaving *flags* set to NULL.

```c
PyObject*
PyRun_StringFlags(const char *str, int start,
PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
```
Execute Python source code from str in the context specified by the objects globals and locals with the compiler flags specified by flags. globals must be a dictionary; locals can be any object that implements the mapping protocol. The parameter start specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or NULL if an exception was raised.

PyObject* PyRun_File(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)

This is a simplified interface to PyRun_FileExFlags() below, leaving closeit set to 0 and flags set to NULL.

PyObject* PyRun_FileEx(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)

This is a simplified interface to PyRun_FileExFlags() below, leaving flags set to NULL.

PyObject* PyRun_FileFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)

This is a simplified interface to PyRun_FileExFlags() below, leaving closeit set to 0.

PyObject* PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)

Similar to PyRun_StringFlags(), but the Python source code is
read from \textit{fp} instead of an in-memory string. \textit{filename} should be the name of the file, it is decoded from the filesystem encoding (\texttt{sys.getfilesystemencoding()}). If \textit{closeit} is true, the file is closed before \texttt{PyRun_FileExFlags()} returns.

\texttt{PyObject* Py\_CompileString}(const char *\textit{str}, const char *\textit{filename}, int \textit{start})

\textit{Return value: New reference.}
This is a simplified interface to \texttt{Py\_CompileStringFlags()} below, leaving \textit{flags} set to \texttt{NULL}.

\texttt{PyObject* Py\_CompileStringFlags}(const char *\textit{str}, const char *\textit{filename}, int \textit{start}, \texttt{PyCompilerFlags*flags})

\textit{Return value: New reference.}
This is a simplified interface to \texttt{Py\_CompileStringExFlags()} below, with \textit{optimize} set to \texttt{-1}.

\texttt{PyObject* Py\_CompileStringObject}(const char *\textit{str}, \texttt{PyObject *filename}, int \textit{start}, \texttt{PyCompilerFlags*flags}, int \textit{optimize})

\textit{Return value: New reference.}
Parse and compile the Python source code in \textit{str}, returning the resulting code object. The start token is given by \textit{start}; this can be used to constrain the code which can be compiled and should be \texttt{Py\_eval\_input}, \texttt{Py\_file\_input}, or \texttt{Py\_single\_input}. The filename specified by \textit{filename} is used to construct the code object and may appear in tracebacks or \texttt{SyntaxError} exception messages. This returns \texttt{NULL} if the code cannot be parsed or compiled.

The integer \textit{optimize} specifies the optimization level of the compiler; a value of \texttt{-1} selects the optimization level of the interpreter as given by \texttt{-0} options. Explicit levels are \texttt{0} (no
optimization; __debug__ is true), 1 (asserts are removed, __debug__ is false) or 2 (docstrings are removed too).

*New in version 3.4.*

`PyObject* Py_CompileStringExFlags(const char *str, const char *filename, int start, PyCompilerFlags *flags, int optimize)`

*Return value: New reference.*

Like `Py_CompileStringObject()`, but `filename` is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

*New in version 3.2.*

`PyObject* PyEval_EvalCode(PyObject *co, PyObject *globals, PyObject *locals)`

*Return value: New reference.*

This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to `NULL`.

`PyObject* PyEval_EvalCodeEx(PyObject *co, PyObject *globals, PyObject *locals, PyObject *const *args, int argcount, PyObject *const *kws, int kwcount, PyObject *const *defs, int defcount, PyObject *kwdefs, PyObject *closure)`

*Return value: New reference.*

Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for `keyword-only` arguments and a closure tuple of cells.

**PyFrameObject**

The C structure of the objects used to describe frame objects. The
fields of this type are subject to change at any time.

PyObject* **PyEval_EvalFrame** (PyFrameObject *f)

*Return value: New reference.*
Evaluate an execution frame. This is a simplified interface to **PyEval_EvalFrameEx()**, for backward compatibility.

PyObject* **PyEval_EvalFrameEx** (PyFrameObject *f, int throwflag)

*Return value: New reference.*
This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame f is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the *throw()* methods of generator objects.

*Changed in version 3.4:* This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

int **PyEval_MergeCompilerFlags** (PyCompilerFlags *cf)
This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int **Py_eval_input**
The start symbol from the Python grammar for isolated expressions; for use with **Py_CompileString()**.

int **Py_file_input**
The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with **Py_CompileString()**. This is the symbol to use when compiling
arbitrarily long Python source code.

```c
int Py_single_input
```

The start symbol from the Python grammar for a single statement; for use with `Py_CompileString()`. This is the symbol used for the interactive interpreter loop.

```c
struct PyCompilerFlags
```

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify `flags`.

Whenever `PyCompilerFlags *flags` is `NULL`, `cf_flags` is treated as equal to 0, and any modification due to `from __future__ import` is discarded.

```c
struct PyCompilerFlags {
    int cf_flags;
}
```

```c
int CO_FUTURE_DIVISION
```

This bit can be set in `flags` to cause division operator `/` to be interpreted as “true division” according to PEP 238.
Reference Counting

The macros in this section are used for managing reference counts of Python objects.

```c
void Py_INCREF(PyObject *o)

Increment the reference count for object o. The object must not be NULL; if you aren't sure that it isn't NULL, use Py_XINCREF().

void Py_XINCREF(PyObject *o)

Increment the reference count for object o. The object may be NULL, in which case the macro has no effect.

void Py_DECREF(PyObject *o)

Decrement the reference count for object o. The object must not be NULL; if you aren't sure that it isn't NULL, use Py_XDECREF(). If the reference count reaches zero, the object's type's deallocation function (which must not be NULL) is invoked.

Warning: The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a __del__() method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before Py_DECREF() is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call Py_DECREF() for the temporary variable.

void Py_XDECREF(PyObject *o)
```
Decrement the reference count for object \( o \). The object may be
\textit{NULL}, in which case the macro has no effect; otherwise the effect
is the same as for \texttt{Py\_DECREF()}, and the same warning applies.

\begin{verbatim}
void \textbf{Py\_CLEAR}(PyObject *o)
\end{verbatim}

Decrement the reference count for object \( o \). The object may be
\textit{NULL}, in which case the macro has no effect; otherwise the effect
is the same as for \texttt{Py\_DECREF()}, except that the argument is also
set to \textit{NULL}. The warning for \texttt{Py\_DECREF()} does not apply with
respect to the object passed because the macro carefully uses a
temporary variable and sets the argument to \textit{NULL} before
decrementing its reference count.

It is a good idea to use this macro whenever decrementing the
value of a variable that might be traversed during garbage
collection.

The following functions are for runtime dynamic embedding of Python:
\texttt{Py\_IncRef(PyObject *o)}, \texttt{Py\_DecRef(PyObject *o)}. They
are simply exported function versions of \texttt{Py\_XINCREF()} and
\texttt{Py\_XDECREF()}, respectively.

The following functions or macros are only for use within the
interpreter core: \_\_Py\_Dealloc(), \_\_Py\_ForgetReference(),
\_\_Py\_NewReference(), as well as the global variable
\_\_Py\_RefTotal.
Exception Handling

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX errno variable: there is a global indicator (per thread) of the last error that occurred. Most C API functions don’t clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually NULL if they are supposed to return a pointer, or -1 if they return an integer (exception: the PyArg_*() functions return 1 for success and 0 for failure).

Concretely, the error indicator consists of three object pointers: the exception’s type, the exception's value, and the traceback object. Any of those pointers can be NULL if non-set (although some combinations are forbidden, for example you can’t have a non-NULL traceback if the exception type is NULL).

When a function must fail because some function it called failed, it generally doesn’t set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should not continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

**Note:** The error indicator is not the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is
therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).
Printing and clearing

void **PyErr_Clear**()

Clear the error indicator. If the error indicator is not set, there is no effect.

void **PyErr_PrintEx**(int **set_sys_last_vars**)

Print a standard traceback to **sys.stderr** and clear the error indicator. **Unless** the error is a **SystemExit**. In that case the no traceback is printed and Python process will exit with the error code specified by the **SystemExit** instance.

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error!

If **set_sys_last_vars** is nonzero, the variables **sys.last_type**, **sys.last_value** and **sys.last_traceback** will be set to the type, value and traceback of the printed exception, respectively.

void **PyErr_Print**()

Alias for **PyErr_PrintEx**(1).

void **PyErr_WriteUnraisable**(PyObject * **obj**)

This utility function prints a warning message to **sys.stderr** when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an **__del__()** method.

The function is called with a single argument **obj** that identifies the context in which the unraisable exception occurred. If possible, the repr of **obj** will be printed in the warning message.
Raising exceptions

These functions help you set the current thread’s error indicator. For convenience, some of these functions will always return a NULL pointer for use in a return statement.

```c
void PyErr_SetString(PyObject *type, const char *message)
```

This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. PyExc_RuntimeError. You need not increment its reference count. The second argument is an error message; it is decoded from 'utf-8'.

```c
void PyErr_SetObject(PyObject *type, PyObject *value)
```

This function is similar to PyErr_SetString() but lets you specify an arbitrary Python object for the “value” of the exception.

```c
PyObject* PyErr_Format(PyObject *exception, const char *format, ...)
```

Return value: Always NULL.

This function sets the error indicator and returns NULL. exception should be a Python exception class. The format and subsequent parameters help format the error message; they have the same meaning and values as in PyUnicode_FromFormat(). format is an ASCII-encoded string.

```c
PyObject* PyErr_FormatV(PyObject *exception, const char *format, va_list vargs)
```

Return value: Always NULL.

Same as PyErr_Format(), but taking a va_list argument rather than a variable number of arguments.
New in version 3.5.

```c
void PyErr_SetNone(PyObject *type)
    This is a shorthand for PyErr_SetObject(type, Py_None).
```

```c
int PyErr_BadArgument()
    This is a shorthand for PyErr_SetString(PyExc_TypeError, message),
    where message indicates that a built-in operation was
    invoked with an illegal argument. It is mostly for internal use.
```

```c
PyObject* PyErr_NoMemory()
    Return value: Always NULL.
    This is a shorthand for PyErr_SetNone(PyExc_MemoryError);
    it returns NULL so an object allocation function can write return
    PyErr_NoMemory(); when it runs out of memory.
```

```c
PyObject* PyErr_SetFromErrno(PyObject *type)
    Return value: Always NULL.
    This is a convenience function to raise an exception when a C
    library function has returned an error and set the C variable
    errno. It constructs a tuple object whose first item is the integer
    errno value and whose second item is the corresponding error
    message (gotten from strerror()), and then calls
    PyErr_SetObject(type, object). On Unix, when the errno
    value is EINTR, indicating an interrupted system call, this calls
    PyErr_CheckSignals(), and if that set the error indicator,
    leaves it set to that. The function always returns NULL, so a
    wrapper function around a system call can write return
    PyErr_SetFromErrno(type); when the system call returns an
    error.
```

```c
PyObject*
```
**PyErr_SetFromErrnoWithFilenameObject** (PyObject *type
PyObject *filenameObject)

*Return value: Always NULL.*
Similar to **PyErr_SetFromErrno()**, with the additional behavior that if *filenameObject* is not NULL, it is passed to the constructor of *type* as a third parameter. In the case of OSError exception, this is used to define the filename attribute of the exception instance.

**PyObject***

**PyErr_SetFromErrnoWithFilenameObjects** (PyObject *type
PyObject *filenameObject, PyObject *filenameObject2)

*Return value: Always NULL.*
Similar to **PyErr_SetFromErrnoWithFilenameObject()**, but takes a second filename object, for raising errors when a function that takes two filenames fails.

*New in version 3.4.*

**PyObject***

**PyErr_SetFromErrnoWithFilename** (PyObject *type, const char *filename)

*Return value: Always NULL.*
Similar to **PyErr_SetFromErrnoWithFilenameObject()**, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (os.fsdecode()).

**PyObject*** **PyErr_SetFromWindowsErr** (int ierr)

*Return value: Always NULL.*
This is a convenience function to raise WindowsError. If called with *ierr* of 0, the error code returned by a call to GetLastError() is used instead. It calls the Win32 function FormatMessage() to retrieve the Windows description of error
code given by *ierr* or GetLastError(), then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from FormatMessage()), and then calls PyErr_SetObject(PyExc_WindowsError, object). This function always returns *NULL*.

**Availability**: Windows.

**PyObject**

*PyErr_SetExcFromWindowsErr*(PyObject *type, int *ierr*)

*Return value: Always NULL.*

Similar to PyErr_SetFromWindowsErr(), with an additional parameter specifying the exception type to be raised.

**Availability**: Windows.

**PyObject**

*PyErr_SetFromWindowsErrWithFilename*(int *ierr*, const char *filename*)

*Return value: Always NULL.*

Similar to PyErr_SetFromWindowsErrWithFilenameObject(), but the filename is given as a C string. *filename* is decoded from the filesystem encoding (os.fsdecode()).

**Availability**: Windows.

**PyObject**

*PyErr_SetExcFromWindowsErrWithFilenameObject*(int *ierr*, PyObject *filename*)

*Return value: Always NULL.*

Similar to
PyErr_SetFromWindowsErrWithFilenameObject(), with an additional parameter specifying the exception type to be raised.

**Availability:** Windows.

```python
PyObject*
PyErr_SetExcFromWindowsErrWithFilenameObjects(  
    int ierr, PyObject *filename, PyObject *filename2)
```

*Return value: Always NULL.*

Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, but accepts a second filename object.

**Availability:** Windows.

*New in version 3.4.*

```python
PyObject*
PyErr_SetExcFromWindowsErrWithFilename(  
    PyObject *type,  
    int ierr, const char *filename)
```

*Return value: Always NULL.*

Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, with an additional parameter specifying the exception type to be raised.

**Availability:** Windows.

```python
PyObject* PyErr_SetImportError(  
    PyObject *msg,  
    PyObject *name, PyObject *path)
```

*Return value: Always NULL.*

This is a convenience function to raise `ImportError`. `msg` will be set as the exception’s message string. `name` and `path`, both of which can be NULL, will be set as the `ImportError`’s respective `name` and `path` attributes.
New in version 3.3.

```c
void PyErr_SyntaxLocationObject(PyObject *filename, int lineno, int col_offset)
```

Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

New in version 3.4.

```c
void PyErr_SyntaxLocationEx(const char *filename, int lineno, int col_offset)
```

Like `PyErr_SyntaxLocationObject()`, but `filename` is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

New in version 3.2.

```c
void PyErr_SyntaxLocation(const char *filename, int lineno)
```

Like `PyErr_SyntaxLocationEx()`, but the `col_offset` parameter is omitted.

```c
void PyErr_BadInternalCall()
```

This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where `message` indicates that an internal operation (e.g., a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.
Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python `warnings` module. They normally print a warning message to `sys.stderr`; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is 0 if no exception is raised, or -1 if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_XDECREF()`) owned references and return an error value.

```c
int PyErr_WarnEx(PyObject *category, const char *message,
Py_ssize_t stack_level)
```

Issue a warning message. The `category` argument is a warning category (see below) or `NULL`; the `message` argument is a UTF-8 encoded string. `stack_level` is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A `stack_level` of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth.

Warning categories must be subclasses of `PyExc_WARNING`; `PyExc_WARNING` is a subclass of `PyExc_Exception`; the default warning category is `PyExc_RuntimeWarning`. The standard Python warning categories are available as global variables whose names are enumerated at `Standard Warning Categories`. 
For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

```c
PyObject*
PyErr_SetImportErrorSubclass(PyObject *msg, PyObject *name, PyObject *path)
```

*Return value: Always NULL.*

Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

*New in version 3.6.*

```c
int PyErr_WarnExplicitObject(PyObject *category, PyObject *message, PyObject *filename, int lineno, PyObject *module, PyObject *registry)
```

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function `warnings.warn_explicit()`, see there for more information. The `module` and `registry` arguments may be set to `NULL` to get the default effect described there.

*New in version 3.4.*

```c
int PyErr_WarnExplicit(PyObject *category, const char *message, const char *filename, int lineno, const char *module, PyObject *registry)
```

Similar to `PyErr_WarnExplicitObject()` except that `message` and `module` are UTF-8 encoded strings, and `filename` is decoded from the filesystem encoding (`os.fsdecode()`).

```c
int PyErr_WarnFormat(PyObject *category, Py_ssize_t stack_level, const char *format, ...)
```
Function similar to `PyErr_WarnEx()`, but use `PyUnicode_FromFormat()` to format the warning message. `format` is an ASCII-encoded string.

*New in version 3.2.*

```c
int PyErr_ResourceWarning(PyObject *source,
                        Py_ssize_t stack_level, const char *format, ...

Function similar to `PyErr_WarnFormat()`, but category is `ResourceWarning` and pass `source` to `warnings.WarningMessage()`.

*New in version 3.6.*
Querying the error indicator

PyObject* **PyErr_Occurred()**

*Return value: Borrowed reference.*

Test whether the error indicator is set. If set, return the exception type (the first argument to the last call to one of the PyErr_Set*() functions or to PyErr_Restore(())). If not set, return NULL. You do not own a reference to the return value, so you do not need to Py_DECREF() it.

**Note:** Do not compare the return value to a specific exception; use PyErr_ExceptionMatches() instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

```c
int PyErr_ExceptionMatches(PyObject *exc)
```

Equivalent to PyErr_GivenExceptionMatches(PyErr_Occurred(), exc). This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

```c
int PyErr_GivenExceptionMatches(PyObject *given,PyObject *exc)
```

Return true if the given exception matches the exception type in exc. If exc is a class object, this also returns true when given is an instance of a subclass. If exc is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

```c
void PyErr_Fetch(PyObject **ptype, PyObject **pvalue,
```
Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to NULL. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be NULL even when the type object is not.

**Note:** This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g.:

```c
{
    PyObject *type, *value, *traceback;
    PyErr_Fetch(&type, &value, &traceback);

    /* ... code that might produce other errors ... */
    PyErr_Restore(type, value, traceback);
}
```

```
void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
```

Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are NULL, the error indicator is cleared. Do not pass a NULL type and non-NULL value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don’t understand this, don’t use this function. I warned you.)
Note: This function is normally only used by code that needs to save and restore the error indicator temporarily. Use PyErr_Fetch() to save the current error indicator.

void PyErr_NormalizeException(PyObject** exc, PyObject** val, PyObject** tb)

Under certain circumstances, the values returned by PyErr_Fetch() below can be “unnormalized”, meaning that *exc is a class object but *val is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

Note: This function does not implicitly set the __traceback__ attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed:

```c
if (tb != NULL) {
    PyException_SetTraceback(val, tb);
}
```

void PyErr_GetExcInfo(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)

Retrieve the exception info, as known from sys.exc_info(). This refers to an exception that was already caught, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be NULL. Does not modify the exception info state.

Note: This function is not normally used by code that wants to
handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_SetExcInfo()` to restore or clear the exception state.

New in version 3.3.

```c
void PyErr_SetExcInfo(PyObject *type, PyObject *value, PyObject *traceback)
```
Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. For general rules about the three arguments, see `PyErr_Restore()`.

**Note:** This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_GetExcInfo()` to read the exception state.

New in version 3.3.
Signal Handling

`int PyErr_CheckSignals()`  
This function interacts with Python’s signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for SIGINT is to raise the `KeyboardInterrupt` exception. If an exception is raised the error indicator is set and the function returns `-1`; otherwise the function returns `0`. The error indicator may or may not be cleared if it was previously set.

`void PyErr_SetInterrupt()`  
This function simulates the effect of a SIGINT signal arriving — the next time `PyErr_CheckSignals()` is called, `KeyboardInterrupt` will be raised. It may be called without holding the interpreter lock.

`int PySignal_SetWakeupFd(int fd)`  
This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. `fd` must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. `fd` should be a valid file descriptor. The function should only be called from the main thread.

*Changed in version 3.5:* On Windows, the function now also supports socket handles.
Exception Classes

PyObject* PyErr_NewException(const char *name, PyObject *base, PyObject *dict)


This utility function creates and returns a new exception class. The name argument must be the name of the new exception, a C string of the form module.classname. The base and dict arguments are normally NULL. This creates a class object derived from Exception (accessible in C as PyExc_Exception).

The __module__ attribute of the new class is set to the first part (up to the last dot) of the name argument, and the class name is set to the last part (after the last dot). The base argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The dict argument can be used to specify a dictionary of class variables and methods.

PyObject* PyErr_NewExceptionWithDoc(const char *name, const char *doc, PyObject *base, PyObject *dict)


Same as PyErr_NewException(), except that the new exception class can easily be given a docstring: If doc is non-NULL, it will be used as the docstring for the exception class.

New in version 3.2.
Exception Objects

**PyObject** PyException_GetTraceback(**PyObject** *ex)

*Return value: New reference.*
Return the traceback associated with the exception as a new reference, as accessible from Python through `__traceback__`. If there is no traceback associated, this returns `NULL`.

**int** PyException_SetTraceback(**PyObject** *ex, **PyObject** *tb)
Set the traceback associated with the exception to `tb`. Use `Py_None` to clear it.

**PyObject** PyException_GetContext(**PyObject** *ex)

*Return value: New reference.*
Return the context (another exception instance during whose handling `ex` was raised) associated with the exception as a new reference, as accessible from Python through `__context__`. If there is no context associated, this returns `NULL`.

**void** PyException_SetContext(**PyObject** *ex, **PyObject** *ctx)
Set the context associated with the exception to `ctx`. Use `NULL` to clear it. There is no type check to make sure that `ctx` is an exception instance. This steals a reference to `ctx`.

**PyObject** PyException_GetCause(**PyObject** *ex)

*Return value: New reference.*
Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

**void** PyException_SetCause(**PyObject** *ex, **PyObject** *cause)
Set the cause associated with the exception to `cause`. Use `NULL` to clear it. There is no type check to make sure that `cause` is either an exception instance or `None`. This steals a reference to `cause`.

`__suppress_context__` is implicitly set to `True` by this function.
Unicode Exception Objects

The following functions are used to create and modify Unicode exceptions from C.

PyObject* PyUnicodeDecodeError_Create(const char *encoding, const char *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a UnicodeDecodeError object with the attributes encoding, object, length, start, end and reason. encoding and reason are UTF-8 encoded strings.

PyObject* PyUnicodeEncodeError_Create(const char *encoding, const Py_UNICODE *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a UnicodeEncodeError object with the attributes encoding, object, length, start, end and reason. encoding and reason are UTF-8 encoded strings.

PyObject* PyUnicodeTranslateError_Create(const Py_UNICODE *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Create a UnicodeTranslateError object with the attributes object, length, start, end and reason. reason is a UTF-8 encoded string.

PyObject*
PyUnicodeDecodeError_GetEncoding(PyObject *exc)
PyObject*
PyUnicodeEncodeError_GetEncoding(PyObject *exc)

Return the encoding attribute of the given exception object.

PyObject *
PyUnicodeDecodeError_GetObject(PyObject *exc)
PyObject *
PyUnicodeEncodeError_GetObject(PyObject *exc)
PyObject *
PyUnicodeTranslateError_GetObject(PyObject *exc)

Return the object attribute of the given exception object.

int PyUnicodeDecodeError_GetStart(PyObject *exc, Py_ssize_t *start)
int PyUnicodeEncodeError_GetStart(PyObject *exc, Py_ssize_t *start)
int PyUnicodeTranslateError_GetStart(PyObject *exc, Py_ssize_t *start)

Get the start attribute of the given exception object and place it into *start. start must not be NULL. Return 0 on success, -1 on failure.

int PyUnicodeDecodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeEncodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeTranslateError_SetStart(PyObject *exc, Py_ssize_t start)

Set the start attribute of the given exception object to start. Return 0 on success, -1 on failure.

int PyUnicodeDecodeError_GetEnd(PyObject *exc, Py_ssize_t *end)
int PyUnicodeEncodeError_GetEnd(PyObject *exc, Py_ssize_t *end)

Get the end attribute of the given exception object and place it into *end. end must not be NULL. Return 0 on success, -1 on failure.

int PyUnicodeTranslateError_GetEnd(PyObject *exc, Py_ssize_t *end)

int PyUnicodeDecodeError_GetEnd(PyObject *exc, Py_ssize_t *end)

Set the end attribute of the given exception object to end. Return 0 on success, -1 on failure.

PyObject*
PyUnicodeDecodeError_GetReason(PyObject *exc)

PyObject*
PyUnicodeEncodeError_GetReason(PyObject *exc)

PyObject*
PyUnicodeTranslateError_GetReason(PyObject *exc)


Return the reason attribute of the given exception object.

int PyUnicodeDecodeError_SetReason(PyObject *exc, const char *reason)

int PyUnicodeEncodeError_SetReason(PyObject *exc, const char *reason)

int PyUnicodeTranslateError_SetReason(PyObject *exc, const char *reason)

Set the reason attribute of the given exception object to reason. Return 0 on success, -1 on failure.
Recursion Control

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically).

```c
int Py_EnterRecursiveCall(const char *where)
```

Marks a point where a recursive C-level call is about to be performed.

If USE_STACKCHECK is defined, this function checks if the OS stack overflowed using `PyOS_CheckStack()`. In this case, it sets a `MemoryError` and returns a nonzero value.

The function then checks if the recursion limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

*where* should be a string such as "in instance check" to be concatenated to the `RecursionError` message caused by the recursion depth limit.

```c
void Py_LeaveRecursiveCall()
```

Ends a `Py_EnterRecursiveCall()`. Must be called once for each `successful` invocation of `Py_EnterRecursiveCall()`.

Properly implementing `tp_repr` for container types requires special recursion handling. In addition to protecting the stack, `tp_repr` also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recurseive_repr()`.
int `Py_ReprEnter(PyObject *object)`

Called at the beginning of the `tp_repr` implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the `tp_repr` implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the `tp_repr` implementation should typically return `NULL`.

Otherwise, the function returns zero and the `tp_repr` implementation can continue normally.

void `Py_ReprLeave(PyObject *object)`

Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.
Standard Exceptions

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

<table>
<thead>
<tr>
<th>C Name</th>
<th>Python Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_BaseException</td>
<td>BaseException</td>
</tr>
<tr>
<td>PyExc_Exception</td>
<td>Exception</td>
</tr>
<tr>
<td>PyExc_ArithmeticError</td>
<td>ArithmeticError</td>
</tr>
<tr>
<td>PyExc_AssertionError</td>
<td>AssertionError</td>
</tr>
<tr>
<td>PyExc_AttributeError</td>
<td>AttributeError</td>
</tr>
<tr>
<td>PyExc_BlockingIOError</td>
<td>BlockingIOError</td>
</tr>
<tr>
<td>PyExc_BrokenPipeError</td>
<td>BrokenPipeError</td>
</tr>
<tr>
<td>PyExc_BufferError</td>
<td>BufferError</td>
</tr>
<tr>
<td>PyExc_ChildProcessError</td>
<td>ChildProcessError</td>
</tr>
<tr>
<td>PyExc_ConnectionAbortedError</td>
<td>ConnectionAbortedError</td>
</tr>
<tr>
<td>PyExc_ConnectionError</td>
<td>ConnectionError</td>
</tr>
<tr>
<td>PyExc_ConnectionRefusedError</td>
<td>ConnectionRefusedError</td>
</tr>
<tr>
<td>PyExc_ConnectionResetError</td>
<td>ConnectionResetError</td>
</tr>
<tr>
<td>PyExc_EOFError</td>
<td>EOFError</td>
</tr>
<tr>
<td>PyExc_FileExistsError</td>
<td>FileExistsError</td>
</tr>
<tr>
<td>PyExc_FileNotFoundError</td>
<td>FileNotFoundError</td>
</tr>
<tr>
<td>PyExc_FloatingPointError</td>
<td>FloatingPointError</td>
</tr>
<tr>
<td>PyExc_GeneratorExit</td>
<td>GeneratorExit</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>PyExc.ImportError</td>
<td>ImportError</td>
</tr>
<tr>
<td>PyExc.IndentationError</td>
<td>IndentationError</td>
</tr>
<tr>
<td>PyExc.IndexError</td>
<td>IndexError</td>
</tr>
<tr>
<td>PyExc.InterruptedError</td>
<td>InterruptedException</td>
</tr>
<tr>
<td>PyExc.IsADirectoryError</td>
<td>IsADirectoryError</td>
</tr>
<tr>
<td>PyExc.KeyError</td>
<td>KeyError</td>
</tr>
<tr>
<td>PyExc.KeyboardInterrupt</td>
<td>KeyboardInterrupt</td>
</tr>
<tr>
<td>PyExc.LookupError</td>
<td>LookupError</td>
</tr>
<tr>
<td>PyExc.MemoryError</td>
<td>MemoryError</td>
</tr>
<tr>
<td>PyExc.ModuleNotFoundError</td>
<td>ModuleNotFoundError</td>
</tr>
<tr>
<td>PyExc.NameError</td>
<td>NameError</td>
</tr>
<tr>
<td>PyExc.NotADirectoryError</td>
<td>NotADirectoryError</td>
</tr>
<tr>
<td>PyExc.NotImplementedError</td>
<td>NotImplementedError</td>
</tr>
<tr>
<td>PyExc.OSError</td>
<td>OSErrror</td>
</tr>
<tr>
<td>PyExc.OverflowError</td>
<td>OverflowError</td>
</tr>
<tr>
<td>PyExc.PermissionError</td>
<td>PermissionError</td>
</tr>
<tr>
<td>PyExc.ProcessLookupError</td>
<td>ProcessLookupError</td>
</tr>
<tr>
<td>PyExc.RecursionError</td>
<td>RecursionError</td>
</tr>
<tr>
<td>PyExc.ReferenceError</td>
<td>ReferenceError</td>
</tr>
<tr>
<td>PyExc.RuntimeError</td>
<td>RuntimeError</td>
</tr>
<tr>
<td>PyExc_StopAsyncIteration</td>
<td>StopAsyncIteration</td>
</tr>
<tr>
<td>PyExc_StopIteration</td>
<td>StopIteration</td>
</tr>
<tr>
<td>PyExc_SyntaxError</td>
<td>SyntaxError</td>
</tr>
<tr>
<td>PyExc_SystemError</td>
<td>SystemError</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PyExc_SystemExit</td>
<td>SystemExit</td>
</tr>
<tr>
<td>PyExc_TabError</td>
<td>TabError</td>
</tr>
<tr>
<td>PyExc_TimeoutError</td>
<td>TimeoutError</td>
</tr>
<tr>
<td>PyExc_TypeError</td>
<td>TypeError</td>
</tr>
<tr>
<td>PyExc_UnboundLocalError</td>
<td>UnboundLocalError</td>
</tr>
<tr>
<td>PyExc_UnicodeDecodeError</td>
<td>UnicodeDecodeError</td>
</tr>
<tr>
<td>PyExc_UnicodeEncodeError</td>
<td>UnicodeEncodeError</td>
</tr>
<tr>
<td>PyExc_UnicodeError</td>
<td>UnicodeError</td>
</tr>
<tr>
<td>PyExc_UnicodeTranslateError</td>
<td>UnicodeTranslateError</td>
</tr>
<tr>
<td>PyExc_ValueError</td>
<td>ValueError</td>
</tr>
<tr>
<td>PyExc_ZeroDivisionError</td>
<td>ZeroDivisionError</td>
</tr>
</tbody>
</table>

*New in version 3.3:* `PyExc_BlockingIOError`, `PyExc_BrokenPipeError`, `PyExc_ChildProcessError`, `PyExc_ConnectionError`, `PyExc_ConnectionAbortedError`, `PyExc_ConnectionRefusedError`, `PyExc_ConnectionResetError`, `PyExc_FileExistsError`, `PyExc_FileNotFoundError`, `PyExc_InterruptedError`, `PyExc_IsADirectoryError`, `PyExc_NotADirectoryError`, `PyExc_PermissionError`, `PyExc_ProcessLookupError` and `PyExc_TimeoutError` were introduced following [PEP 3151](https://www.python.org/dev/peps/pep-03151/).

*New in version 3.5:* `PyExc_StopAsyncIteration` and `PyExc_RecursionError`.

*New in version 3.6:* `PyExc_ModuleNotFoundError`.
These are compatibility aliases to PyExc_OSError:

<table>
<thead>
<tr>
<th>C Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_EnvironmentError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IOError</td>
<td></td>
</tr>
<tr>
<td>PyExc_WindowsError</td>
<td>(3)</td>
</tr>
</tbody>
</table>

*Changed in version 3.3:* These aliases used to be separate exception types.

Notes:

1. This is a base class for other standard exceptions.
2. This is the same as `weakref.ReferenceError`.
3. Only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.
Standard Warning Categories

All standard Python warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

<table>
<thead>
<tr>
<th>C Name</th>
<th>Python Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_Warning</td>
<td>Warning</td>
</tr>
<tr>
<td>PyExc_BytesWarning</td>
<td>BytesWarning</td>
</tr>
<tr>
<td>PyExc_DeprecationWarning</td>
<td>DeprecationWarning</td>
</tr>
<tr>
<td>PyExc_FutureWarning</td>
<td>FutureWarning</td>
</tr>
<tr>
<td>PyExc_ImportWarning</td>
<td>ImportWarning</td>
</tr>
<tr>
<td>PyExc_PendingDeprecationWarning</td>
<td>PendingDeprecationWarning</td>
</tr>
<tr>
<td>PyExc_ResourceWarning</td>
<td>ResourceWarning</td>
</tr>
<tr>
<td>PyExc_RuntimeWarning</td>
<td>RuntimeWarning</td>
</tr>
<tr>
<td>PyExc_SyntaxWarning</td>
<td>SyntaxWarning</td>
</tr>
<tr>
<td>PyExc_UnicodeWarning</td>
<td>UnicodeWarning</td>
</tr>
<tr>
<td>PyExc_UserWarning</td>
<td>UserWarning</td>
</tr>
</tbody>
</table>

*New in version 3.2:* `PyExc_ResourceWarning`.

Notes:

1. This is a base class for other standard warning categories.
Utilities

The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

- Operating System Utilities
- System Functions
- Process Control
- Importing Modules
- Data marshalling support
- Parsing arguments and building values
  - Parsing arguments
    - Strings and buffers
    - Numbers
    - Other objects
    - API Functions
  - Building values
- String conversion and formatting
- Reflection
- Codec registry and support functions
  - Codec lookup API
  - Registry API for Unicode encoding error handlers
Operating System Utilities

PyObject* **PyOS_FSPath**(PyObject *path)

*Return value: New reference.*

Return the file system representation for *path*. If the object is a *str* or *bytes* object, then its reference count is incremented. If the object implements the *os.PathLike* interface, then __fspath__() is returned as long as it is a *str* or *bytes* object. Otherwise *TypeError* is raised and NULL is returned.

*New in version 3.6.*

int **Py_FdIsInteractive**(FILE *fp, const char *filename)

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which *isatty(fileno(fp))* is true. If the global flag *Py_InteractiveFlag* is true, this function also returns true if the *filename* pointer is NULL or if the name is equal to one of the strings 'stdin' or '???'.

void **PyOS_BeforeFork**()

Function to prepare some internal state before a process fork. This should be called before calling *fork()* or any similar function that clones the current process. Only available on systems where *fork()* is defined.

*New in version 3.7.*

void **PyOS_AfterFork_Parent**()

Function to update some internal state after a process fork. This should be called from the parent process after calling *fork()* or
any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where fork() is defined.

*New in version 3.7.*

```c
void PyOS_AfterFork_Child()
```

Function to update internal interpreter state after a process fork. This must be called from the child process after calling fork(), or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where fork() is defined.

*New in version 3.7.*

**See also:** os.register_at_fork() allows registering custom Python functions to be called by PyOS_BeforeFork(), PyOS_AfterFork_Parent() and PyOS_AfterFork_Child().

```c
void PyOS_AfterFork()
```

Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

*Deprecated since version 3.7:* This function is superseded by PyOS_AfterFork_Child().

```c
int PyOS_CheckStack()
```

Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when USE_STACKCHECK is
defined (currently on Windows using the Microsoft Visual C++
compiler). USE_STACKCHECK will be defined automatically; you
should never change the definition in your own code.

PyOS_sighandler_t PyOS_getsig(int i)

Return the current signal handler for signal i. This is a thin wrapper
around either sigaction() or signal(). Do not call those
functions directly! PyOS_sighandler_t is a typedef alias for
void (*)(int).

PyOS_sighandler_t PyOS_setsig(int i, PyOS_sighandler_t h)

Set the signal handler for signal i to be h; return the old signal
handler. This is a thin wrapper around either sigaction() or
signal(). Do not call those functions directly! PyOS_sighandler_t is a typedef alias for void (*)(int).

wchar_t* Py_DecodeLocale(const char* arg, size_t *size)

Decode a byte string from the locale encoding with the
surrogateescape error handler: undecodable bytes are decoded as
characters in range U+DC80..U+DCFF. If a byte sequence can be
decoded as a surrogate character, escape the bytes using the
surrogateescape error handler instead of decoding them.

Encoding, highest priority to lowest priority:

- **UTF-8** on macOS and Android;
- **UTF-8** if the Python UTF-8 mode is enabled;
- **ASCII** if the LC_CTYPE locale is "C", nl_langinfo(CODESET) returns the ASCII encoding (or an alias), and mbstowcs() and wcstombs() functions uses the ISO-8859-1 encoding.
- the current locale encoding.
Return a pointer to a newly allocated wide character string, use `PyMem_RawFree()` to free the memory. If size is not NULL, write the number of wide characters excluding the null character into *size.

Return NULL on decoding error or memory allocation error. If size is not NULL, *size is set to (size_t)-1 on memory error or set to (size_t)-2 on decoding error.

Decoding errors should never happen, unless there is a bug in the C library.

Use the `Py_EncodeLocale()` function to encode the character string back to a byte string.

**See also:** The `PyUnicode_DecodeFSDefaultAndSize()` and `PyUnicode_DecodeLocaleAndSize()` functions.

*New in version 3.5.*

*Changed in version 3.7:* The function now uses the UTF-8 encoding in the UTF-8 mode.

```c
char* Py_EncodeLocale(const wchar_t *text, size_t *error_pos)
```

Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- UTF-8 on macOS and Android;
- UTF-8 if the Python UTF-8 mode is enabled;
- ASCII if the LC_CTYPE locale is "C",
\texttt{nl_langinfo(CODESET)} returns the ASCII encoding (or an alias), and \texttt{mbstowcs()} and \texttt{wcstombs()} functions uses the ISO-8859-1 encoding.

- the current locale encoding.

The function uses the UTF-8 encoding in the Python UTF-8 mode.

Return a pointer to a newly allocated byte string, use \texttt{PyMem_Free()} to free the memory. Return \texttt{NULL} on encoding error or memory allocation error.

If error\_pos is not NULL, \*error\_pos is set to (size\_t)-1 on success, or set to the index of the invalid character on encoding error.

Use the \texttt{Py_DecodeLocale()} function to decode the bytes string back to a wide character string.

\textit{Changed in version 3.7}: The function now uses the UTF-8 encoding in the UTF-8 mode.

\begin{tabular}{|l|}
\hline
\textbf{See also:} The \texttt{PyUnicode_EncodeFSDefault()} and \texttt{PyUnicode_EncodeLocale()} functions. \\
\hline
\end{tabular}

\textit{New in version 3.5}.

\textit{Changed in version 3.7}: The function now supports the UTF-8 mode.
System Functions

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread's `sys` module's dict, which is contained in the internal thread state structure.

**PyObject \*PySys_GetObject(const char \*name)**

*Return value: Borrowed reference.*
Return the object `name` from the `sys` module or `NULL` if it does not exist, without setting an exception.

**int PySys_SetObject(const char \*name, PyObject \*v)**

Set `name` in the `sys` module to `v` unless `v` is `NULL`, in which case `name` is deleted from the `sys` module. Returns `0` on success, `-1` on error.

**void PySys_ResetWarnOptions()**

Reset `sys.warnoptions` to an empty list. This function may be called prior to `Py_Initialize()`.

**void PySys_AddWarnOption(const wchar_t \*s)**

Append `s` to `sys.warnoptions`. This function must be called prior to `Py_Initialize()` in order to affect the warnings filter list.

**void PySys_AddWarnOptionUnicode(PyObject \*unicode)**

Append `unicode` to `sys.warnoptions`.

Note: this function is not currently usable from outside the CPython implementation, as it must be called prior to the implicit import of
warnings in Py_Initialize() to be effective, but can’t be called until enough of the runtime has been initialized to permit the creation of Unicode objects.

void PySys_SetPath(const wchar_t *path)
Set sys.path to a list object of paths found in path which should be a list of paths separated with the platform’s search path delimiter (: on Unix, ; on Windows).

void PySys_WriteStdout(const char *format, ...)
Write the output string described by format to sys.stdout. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted “%s” formats should occur; these should be limited using “%.<N>s” where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for “%f”, which can print hundreds of digits for very large numbers.

If a problem occurs, or sys.stdout is unset, the formatted message is written to the real (C level) stdout.

void PySys_WriteStderr(const char *format, ...)
As PySys_WriteStdout(), but write to sys.stderr instead.

void PySys_FormatStdout(const char *format, ...)
Function similar to PySys_WriteStdout() but format the message using PyUnicode_FromFormatV() and don’t truncate the
message to an arbitrary length.

*New in version 3.2.*

```c
void PySys_FormatStderr(const char *format, ...)
```
As `PySys_FormatStdout()`, but write to `sys.stderr` or `stderr` instead.

*New in version 3.2.*

```c
void PySys_AddXOption(const wchar_t *s)
```
Parse `s` as a set of `-X` options and add them to the current options mapping as returned by `PySys_GetXOptions()`. This function may be called prior to `Py_Initialize()`.

*New in version 3.2.*

```c
PyObject *PySys_GetXOptions()
```
*Return value: Borrowed reference.*
Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

*New in version 3.2.*
Process Control

void **Py_FatalError**(const char *message)

Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function abort() is called which will attempt to produce a core file.

void **Py_Exit**(int status)

Exit the current process. This calls **Py_FinalizeEx()** and then calls the standard C library function exit(status). If **Py_FinalizeEx()** indicates an error, the exit status is set to 120.

*Changed in version 3.6:* Errors from finalization no longer ignored.

int **Py_AtExit**(void (*func)())

Register a cleanup function to be called by **Py_FinalizeEx()**. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, **Py_AtExit()** returns 0; on failure, it returns -1. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python’s internal finalization will have completed before the cleanup function, no Python APIs should be called by *func.*
The Python Software Foundation is a non-profit corporation. Please donate.

Last updated on Mar 25, 2019. Found a bug?

Created using Sphinx 1.8.5.
Importing Modules

PyObject* **PyImport_ImportModule**(const char *name)

*Return value: New reference.*

This is a simplified interface to `PyImport_ImportModuleEx()` below, leaving the `globals` and `locals` arguments set to `NULL` and `level` set to 0. When the `name` argument contains a dot (when it specifies a submodule of a package), the `fromlist` argument is set to the list `["*"]` so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when `name` in fact specifies a subpackage instead of a submodule: the submodules specified in the package’s `__all__` variable are loaded.) Return a new reference to the imported module, or `NULL` with an exception set on failure. A failing import of a module doesn’t leave the module in `sys.modules`.

This function always uses absolute imports.

PyObject* **PyImport_ImportModuleNoBlock**(const char *name)

*Return value: New reference.*

This function is a deprecated alias of `PyImport_ImportModule()`.

*Changed in version 3.3:* This function used to fail immediately when the import lock was held by another thread. In Python 3.3 though, the locking scheme switched to per-module locks for most purposes, so this function’s special behaviour isn’t needed anymore.
PyObject* PyImport_ImportModuleEx(const char *name,
PyObject *globals, PyObject *locals, PyObject *fromlist)

*Return value: New reference.*

Import a module. This is best described by referring to the built-in
Python function __import__().

The return value is a new reference to the imported module or top-
level package, or NULL with an exception set on failure. Like for
__import__(), the return value when a submodule of a package
was requested is normally the top-level package, unless a non-
empty fromlist was given.

Failing imports remove incomplete module objects, like with
PyImport_ImportModule().

PyObject*
PyImport_ImportModuleLevelObject(PyObject *name,
PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

*Return value: New reference.*

Import a module. This is best described by referring to the built-in
Python function __import__(), as the standard __import__() function calls this function directly.

The return value is a new reference to the imported module or top-
level package, or NULL with an exception set on failure. Like for
__import__(), the return value when a submodule of a package
was requested is normally the top-level package, unless a non-
empty fromlist was given.

*New in version 3.3.*

PyObject* PyImport_ImportModuleLevel(const char *name,
PyObject *globals, PyObject *locals, PyObject *fromlist, int level)
Similar to PyImport_ImportModuleLevelObject(), but the name is a UTF-8 encoded string instead of a Unicode object.

Changed in version 3.3: Negative values for level are no longer accepted.

PyObject* PyImport_Import(PyObject *name)

This is a higher-level interface that calls the current “import hook function” (with an explicit level of 0, meaning absolute import). It invokes the __import__() function from the __builtins__ of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

PyObject* PyImport_ReloadModule(PyObject *m)

Reload a module. Return a new reference to the reloaded module, or NULL with an exception set on failure (the module still exists in this case).

PyObject* PyImport_AddModule(PyObject *name)

Return value: Borrowed reference.
Return the module object corresponding to a module name. The name argument may be of the form package.module. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.

Note: This function does not load or import the module; if the module wasn’t already loaded, you will get an empty module
object. Use `PyImport_ImportModule()` or one of its variants to import a module. Package structures implied by a dotted name for `name` are not created if not already present.

*New in version 3.3.*

```c
PyObject* PyImport_AddModule(const char *name)
```

*Return value: Borrowed reference.*

Similar to `PyImport_AddModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

```c
PyObject* PyImport_ExecCodeModule(const char *name, PyObject *co)
```

*Return value: New reference.*

Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error occurred. `name` is removed from `sys.modules` in error cases, even if `name` was already in `sys.modules` on entry to `PyImport_ExecCodeModule()`. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author’s intents) state.

The module’s `__spec__` and `__loader__` will be set, if not set already, with the appropriate values. The spec’s loader will be set to the module’s `__loader__` (if set) and to an instance of `SourceFileLoader` otherwise.

The module’s `__file__` attribute will be set to the code object’s
co_filename. If applicable, __cached__ will also be set.

This function will reload the module if it was already imported. See \texttt{PyImportReloadModule()} for the intended way to reload a module.

If \textit{name} points to a dotted name of the form \texttt{package.module}, any package structures not already created will still not be created.

See also \texttt{PyImportExecCodeModuleEx()} and \texttt{PyImportExecCodeModuleWithPathnames()}.

\begin{verbatim}
PyObject* PyImportExecCodeModuleEx(const char *name, PyObject *co, const char *pathname)
    Like \texttt{PyImportExecCodeModule()}, but the \texttt{__file__} attribute of the module object is set to \texttt{pathname} if it is non-NULL.
    See also \texttt{PyImportExecCodeModuleWithPathnames()}.  

PyObject* PyImportExecCodeModuleObject(PyObject *name, PyObject *co, PyObject *pathname, PyObject *cpathname)
    Like \texttt{PyImportExecCodeModuleEx()}, but the \texttt{__cached__} attribute of the module object is set to \texttt{cpathname} if it is non-NULL.
    Of the three functions, this is the preferred one to use.

    \textit{New in version 3.3.}

PyObject* PyImportExecCodeModuleWithPathnames(const char *name, PyObject *co, const char *pathname, const char *cpathname)
\end{verbatim}
Like `PyImport_ExecCodeModuleObject()`, but `name`, `pathname` and `cpathname` are UTF-8 encoded strings. Attempts are also made to figure out what the value for `pathname` should be from `cpathname` if the former is set to `NULL`.

New in version 3.2.

Changed in version 3.3: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

```c
long PyImport_GetMagicNumber()
```

Return the magic number for Python bytecode files (a.k.a. `.pyc` file). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order. Returns `-1` on error.

Changed in version 3.3: Return value of `-1` upon failure.

```c
const char *PyImport_GetMagicTag()
```

Return the magic tag string for PEP 3147 format Python bytecode file names. Keep in mind that the value at `sys.implementation.cache_tag` is authoritative and should be used instead of this function.

New in version 3.2.

```c
PyObject* PyImport_GetModuleDict()
```

Return value: Borrowed reference.
Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

```c
PyObject* PyImport_GetModule(PyObject *name)
```

Return the already imported module with the given name. If the module has not been imported yet then returns NULL but does not set an error. Returns NULL and sets an error if the lookup failed.

*New in version 3.7.*

**PyObject** `PyImport_GetImporter(PyObject *path)`

*Return value: New reference.*

Return a finder object for a `sys.path/pkg.__path__` item `path`, possibly by fetching it from the `sys.path_importer_cache` dict. If it wasn’t yet cached, traverse `sys.path_hooks` until a hook is found that can handle the path item. Return `None` if no hook could; this tells our caller that the path based finder could not find a finder for this path item. Cache the result in `sys.path_importer_cache`. Return a new reference to the finder object.

```c
void _PyImport_Init()
```

Initialize the import mechanism. For internal use only.

```c
void PyImport_Cleanup()
```

Empty the module table. For internal use only.

```c
void _PyImport_Fini()
```

Finalize the import mechanism. For internal use only.

```c
int PyImport_ImportFrozenModuleObject(PyObject *name)
```

*Return value: New reference.*

Load a frozen module named `name`. Return 1 for success, 0 if the module is not found, and -1 with an exception set if the initialization failed. To access the imported module on a successful load, use `PyImport_ImportModule()`. (Note the misnomer —
this function would reload the module if it was already imported.)

*New in version 3.3.*

*Changed in version 3.4:* The **__file__** attribute is no longer set on the module.

```c
int PyImport_ImportFrozenModule(const char *name)
```

Similar to `PyImport_ImportFrozenModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

```c
def _frozen
```

This is the structure type definition for frozen module descriptors, as generated by the `freeze` utility (see `Tools/freeze/` in the Python source distribution). Its definition, found in `Include/import.h`, is:

```c
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
};
```

```c
const struct _frozen* PyImport_FrozenModules
```

This pointer is initialized to point to an array of `struct _frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

```
int PyImport_AppendInittab(const char *name, PyObject* (*initfunc)(void))
```

Add a single module to the existing table of built-in modules. This is
a convenience wrapper around `PyImport_ExtendInittab()`
returning -1 if the table could not be extended. The new module
can be imported by the name `name`, and uses the function `initfunc`
as the initialization function called on the first attempted import.
This should be called before `Py_Initialize()`.

```c
struct _inittab
    Structure describing a single entry in the list of built-in modules.
    Each of these structures gives the name and initialization function
    for a module built into the interpreter. The name is an ASCII
    encoded string. Programs which embed Python may use an array
    of these structures in conjunction with `PyImport_ExtendInittab()` to provide additional built-in
    modules. The structure is defined in `Include/import.h` as:

    ```c
    struct _inittab {
        const char *name;  /* ASCII encoded */
        PyObject* (*initfunc)(void);
    };
    ```

int `PyImport_ExtendInittab`(struct _inittab *newtab)
    Add a collection of modules to the table of built-in modules. The
    `newtab` array must end with a sentinel entry which contains `NULL`
    for the name field; failure to provide the sentinel value can result in
    a memory fault. Returns 0 on success or -1 if insufficient memory
    could be allocated to extend the internal table. In the event of
    failure, no modules are added to the internal table. This should be
called before `Py_Initialize()`.
Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 shares interned strings in the file, and upon unmarshalling. Version 2 uses a binary format for floating point numbers. `PyMarshal_VERSION` indicates the current file format (currently 2).

```c
void PyMarshal_WriteLongToFile(long value, FILE *file, int version)
    Marshal a long integer, value, to file. This will only write the least-significant 32 bits of value; regardless of the size of the native long type. version indicates the file format.

void PyMarshal_WriteObjectToFile(PyObject *value, FILE *file, int version)
    Marshal a Python object, value, to file. version indicates the file format.

PyObject*
PyMarshal_WriteObjectToString(PyObject *value, int version)
    Return a bytes object containing the marshalled representation of
value. version indicates the file format.

The following functions allow marshalled values to be read back in.

long **PyMarshal_ReadLongFromFile** (FILE *file)

Return a C long from the data stream in a FILE* opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of long.

On error, sets the appropriate exception (**EOFError**) and returns -1.

int **PyMarshal_ReadShortFromFile** (FILE *file)

Return a C short from the data stream in a FILE* opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of short.

On error, sets the appropriate exception (**EOFError**) and returns -1.

**PyObject** PyMarshal_ReadObjectFromFile (FILE *file)

*Return value: New reference.*

Return a Python object from the data stream in a FILE* opened for reading.

On error, sets the appropriate exception (**EOFError**, **ValueError** or **TypeError**) and returns **NULL**.

**PyObject** PyMarshal_ReadLastObjectFromFile (FILE *file)

*Return value: New reference.*

Return a Python object from the data stream in a FILE* opened for reading. Unlike **PyMarshal_ReadObjectFromFile()**, this
function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the deserialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won’t be reading anything else from the file.

On error, sets the appropriate exception (EOFError, ValueError or TypeError) and returns NULL.

PyObject* PyMarshal_ReadObjectFromString(const char *data, Py_ssize_t len)

Return a Python object from the data stream in a byte buffer containing len bytes pointed to by data.

On error, sets the appropriate exception (EOFError, ValueError or TypeError) and returns NULL.
Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in *Extending and Embedding the Python Interpreter*.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.
Parsing arguments

A format string consists of zero or more “format units.” A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

Strings and buffers

These formats allow accessing an object as a contiguous chunk of memory. You don’t have to provide raw storage for the returned unicode or bytes area.

In general, when a format sets a pointer to a buffer, the buffer is managed by the corresponding Python object, and the buffer shares the lifetime of this object. You won’t have to release any memory yourself. The only exceptions are es, es#, et and et#.

However, when a Py_buffer structure gets filled, the underlying buffer is locked so that the caller can subsequently use the buffer even inside a Py_BEGIN_ALLOW_THREADS block without the risk of mutable data being resized or destroyed. As a result, you have to call PyBuffer_Release() after you have finished processing the data (or in any early abort case).

Unless otherwise stated, buffers are not NUL-terminated.
Some formats require a read-only bytes-like object, and set a pointer instead of a buffer structure. They work by checking that the object's `PyBufferProcs.bf_releasebuffer` field is `NULL`, which disallows mutable objects such as `bytearray`.

**Note:** For all # variants of formats (s#, y#, etc.), the type of the length argument (int or Py_ssize_t) is controlled by defining the macro `PY_SSIZE_T_CLEAN` before including `Python.h`. If the macro was defined, length is a Py_ssize_t rather than an int. This behavior will change in a future Python version to only support Py_ssize_t and drop int support. It is best to always define `PY_SSIZE_T_CLEAN`.

`s (str) [const char *]`

Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using 'utf-8' encoding. If this conversion fails, a `UnicodeError` is raised.

**Note:** This format does not accept bytes-like objects. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the 0& format with `PyUnicode_FSConverter()` as converter.

*Changed in version 3.5:* Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s* (str or bytes-like object) [Py_buffer]`
This format accepts Unicode objects as well as bytes-like objects. It fills a `Py_buffer` structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using 'utf-8' encoding.

### s# (str, read-only bytes-like object) [const char *, int or Py_ssize_t]

Like `s*`, except that it doesn’t accept mutable objects. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using 'utf-8' encoding.

### z (str or None) [const char *]

Like `s`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

### z* (str, bytes-like object or None) [Py_buffer]

Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the `Py_buffer` structure is set to `NULL`.

### z# (str, read-only bytes-like object or None) [const char *, int]

Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

### y (read-only bytes-like object) [const char *]

This format converts a bytes-like object to a C pointer to a character string; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes; if it does, a `ValueError` exception is raised.

*Changed in version 3.5:* Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes buffer.

### y* (bytes-like object) [Py_buffer]

This variant on `s*` doesn’t accept Unicode objects, only bytes-like
objects. **This is the recommended way to accept binary data.**

*y# (read-only bytes-like object) [const char *, int]*  
This variant on s# doesn’t accept Unicode objects, only bytes-like objects.

*S (bytes) [PyBytesObject *]*  
Requires that the Python object is a bytes object, without attempting any conversion. Raises TypeError if the object is not a bytes object. The C variable may also be declared as PyObject*.

*Y (bytearray) [PyByteArrayObject *]*  
Requires that the Python object is a bytearray object, without attempting any conversion. Raises TypeError if the object is not a bytearray object. The C variable may also be declared as PyObject*.

*u (str) [const Py_UNICODE *]*  
Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a Py_UNICODE pointer variable, which will be filled with the pointer to an existing Unicode buffer. Please note that the width of a Py_UNICODE character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points; if it does, a ValueError exception is raised.

*Changed in version 3.5:* Previously, TypeError was raised when embedded null code points were encountered in the Python string.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsWideCharString().

*u# (str) [const Py_UNICODE *, int]*
This variant on \texttt{u} stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. This variant allows null code points.

\textit{Deprecated since version 3.3, will be removed in version 4.0:} Part of the old-style \texttt{Py_UNICODE} API; please migrate to using \texttt{PyUnicode_AsWideCharString()}. 

\texttt{Z(\texttt{str} or \texttt{None})[\texttt{const Py_UNICODE *}]} 
Like \texttt{u}, but the Python object may also be \texttt{None}, in which case the \texttt{Py_UNICODE} pointer is set to \texttt{NULL}.

\textit{Deprecated since version 3.3, will be removed in version 4.0:} Part of the old-style \texttt{Py_UNICODE} API; please migrate to using \texttt{PyUnicode_AsWideCharString()}. 

\texttt{Z#(\texttt{str} or \texttt{None})[\texttt{const Py_UNICODE *}, \texttt{int}]} 
Like \texttt{u#}, but the Python object may also be \texttt{None}, in which case the \texttt{Py_UNICODE} pointer is set to \texttt{NULL}.

\textit{Deprecated since version 3.3, will be removed in version 4.0:} Part of the old-style \texttt{Py_UNICODE} API; please migrate to using \texttt{PyUnicode_AsWideCharString()}. 

\texttt{U(\texttt{str})[\texttt{PyObject *}]} 
Requires that the Python object is a Unicode object, without attempting any conversion. Raises \texttt{TypeError} if the object is not a Unicode object. The C variable may also be declared as \texttt{PyObject*}. 

\texttt{w* (read-write \texttt{bytes-like object}) [\texttt{Py_buffer}]} 
This format accepts any object which implements the read-write buffer interface. It fills a \texttt{Py_buffer} structure provided by the caller. The buffer may contain embedded null bytes. The caller
have to call `PyBuffer_Release()` when it is done with the buffer.

**es (str)** [const char *encoding, char **buffer]
This variant on **s** is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case 'utf-8' encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

`PyArg_ParseTuple()` will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust *buffer to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

**et (str, bytes or bytearray)** [const char *encoding, char **buffer]
Same as **es** except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

**es# (str)** [const char *encoding, char **buffer, int *buffer_length]
This variant on **s#** is used for encoding Unicode into a character buffer. Unlike the **es** format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding
as a NUL-terminated string, or NULL, in which case \texttt{utf-8} encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

There are two modes of operation:

If \texttt{*buffer} points a NULL pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set \texttt{*buffer} to reference the newly allocated storage. The caller is responsible for calling \texttt{PyMem_Free()} to free the allocated buffer after usage.

If \texttt{*buffer} points to a non-NULL pointer (an already allocated buffer), \texttt{PyArg_ParseTuple()} will use this location as the buffer and interpret the initial value of \texttt{*buffer\_length} as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a \texttt{ValueError} will be set.

In both cases, \texttt{*buffer\_length} is set to the length of the encoded data without the trailing NUL byte.

\begin{verbatim}

et\# (str, bytes or bytearray) [const char *encoding, char **buffer, int *buffer\_length]

Same as es\# except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Numbers

\end{verbatim}
b (int) [unsigned char]
   Convert a nonnegative Python integer to an unsigned tiny int, stored in a C unsigned char.

B (int) [unsigned char]
   Convert a Python integer to a tiny int without overflow checking, stored in a C unsigned char.

h (int) [short int]
   Convert a Python integer to a C short int.

H (int) [unsigned short int]
   Convert a Python integer to a C unsigned short int, without overflow checking.

i (int) [int]
   Convert a Python integer to a plain C int.

I (int) [unsigned int]
   Convert a Python integer to a C unsigned int, without overflow checking.

l (int) [long int]
   Convert a Python integer to a C long int.

k (int) [unsigned long]
   Convert a Python integer to a C unsigned long without overflow checking.

L (int) [long long]
   Convert a Python integer to a C long long.

K (int) [unsigned long long]
   Convert a Python integer to a C unsigned long long without overflow checking.

n (int) [Py_ssize_t]
   Convert a Python integer to a C Py_ssize_t.
c (bytes or bytearray of length 1) [char]
   Convert a Python byte, represented as a bytes or bytearray object of length 1, to a C char.

   Changed in version 3.3: Allow bytearray objects.

C (str of length 1) [int]
   Convert a Python character, represented as a str object of length 1, to a C int.

f (float) [float]
   Convert a Python floating point number to a C float.

d (float) [double]
   Convert a Python floating point number to a C double.

D (complex) [Py_complex]
   Convert a Python complex number to a C Py_complex structure.

Other objects

O (object) [PyObject *]
   Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object's reference count is not increased. The pointer stored is not NULL.

O! (object) [typeobject, PyObject *]
   Store a Python object in a C object pointer. This is similar to O, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type PyObject*) into which the object pointer is stored. If the Python object does not have the required type, TypeError is raised.

O& (object) [converter, anything]
Convert a Python object to a C variable through a converter function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to void *. The converter function in turn is called as follows:

```
status = converter(object, address);
```

where object is the Python object to be converted and address is the void* argument that was passed to the PyArg_Parse*() function. The returned status should be 1 for a successful conversion and 0 if the conversion has failed. When the conversion fails, the converter function should raise an exception and leave the content of address unmodified.

If the converter returns Py_CLEANUP_SUPPORTED, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the object parameter will be NULL; address will have the same value as in the original call.

*Changed in version 3.1:* Py_CLEANUP_SUPPORTED was added.

**p (bool) [int]**
Tests the value passed in for truth (a boolean predicate) and converts the result to its equivalent C true/false integer value. Sets the int to 1 if the expression was true and 0 if it was false. This accepts any valid Python value. See Truth Value Testing for more information about how Python tests values for truth.

*New in version 3.3.*

**(items) (tuple) [matching-items]**
The object must be a Python sequence whose length is the
number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

It is possible to pass “long” integers (integers whose value exceeds the platform’s LONG_MAX) however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

| Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

$ `PyArg_ParseTupleAndKeywords()` only: Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so | must always be specified before $ in the format string.

New in version 3.3.

; The list of format units ends here; the string after the colon is used as the function name in error messages (the “associated value” of the exception that `PyArg_ParseTuple()` raises).
The list of format units ends here; the string after the semicolon is used as the error message instead of the default error message. ; and ; mutually exclude each other.

Note that any Python object references which are provided to the caller are borrowed references; do not decrement their reference count!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

For the conversion to succeed, the arg object must match the format and the format must be exhausted. On success, the PyArg_Parse*() functions return true, otherwise they return false and raise an appropriate exception. When the PyArg_Parse*() functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

API Functions

int PyArg_ParseTuple(PyObject *args, const char *format, ...)  
Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int PyArg_VaParse(PyObject *args, const char *format, va_list vars)
Identical to `PyArg_ParseTuple()`, except that it accepts a va_list rather than a variable number of arguments.

```c
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], ...)```

Parse the parameters of a function that takes both positional and keyword parameters into local variables. The `keywords` argument is a `NULL`-terminated array of keyword parameter names. Empty names denote positional-only parameters. Returns true on success; on failure, it returns false and raises the appropriate exception.

*Changed in version 3.6:* Added support for positional-only parameters.

```c
int PyArg_VaParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *keywords[], va_list vargs)```

Identical to `PyArg_ParseTupleAndKeywords()`, except that it accepts a va_list rather than a variable number of arguments.

```c
int PyArg_ValidateKeywordArguments(PyObject *)```

Ensure that the keys in the keywords argument dictionary are strings. This is only needed if `PyArg_ParseTupleAndKeywords()` is not used, since the latter already does this check.

*New in version 3.2.*

```c
int PyArg_Parse(PyObject *args, const char *format, ...)```

Function used to deconstruct the argument lists of “old-style” functions — these are functions which use the METH_OLDARGS parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter parsing in new...
code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

```c
int PyArg_UnpackTuple(PyObject *args, const char *name,
Py_ssize_t min, Py_ssize_t max, ...)  
```

A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as `args`; it must actually be a tuple. The length of the tuple must be at least `min` and no more than `max`; `min` and `max` may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject`* variable; these will be filled in with the values from `args`; they will contain borrowed references. The variables which correspond to optional parameters not given by `args` will not be filled in; these should be initialized by the caller. This function returns true on success and false if `args` is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references:

```c
static PyObject *  
weakref_ref(PyObject *self, PyObject *args)  
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;
```
The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()`:

```c
PyArg_ParseTuple(args, "O|O:ref", &object, &callback);
```
Building values

PyObject* Py_BuildValue(const char *format, ...)

*Return value: New reference.*

Create a new value based on a format string similar to those accepted by the PyArg_Parse*() family of functions and a sequence of values. Returns the value or NULL in the case of an error; an exception will be raised if NULL is returned.

Py_BuildValue() does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns None; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the s and s# formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by Py_BuildValue(). In other words, if your code invokes malloc() and passes the allocated memory to Py_BUILDVALUE(), your code is responsible for calling free() for that memory once Py_BUILDVALUE() returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as s#). This can be used
to make long format strings a tad more readable.

s (str or None) [const char *]
Convert a null-terminated C string to a Python str object using 'utf-8' encoding. If the C string pointer is NULL, None is used.

s# (str or None) [const char *, int]
Convert a C string and its length to a Python str object using 'utf-8' encoding. If the C string pointer is NULL, the length is ignored and None is returned.

y (bytes) [const char *]
This converts a C string to a Python bytes object. If the C string pointer is NULL, None is returned.

y# (bytes) [const char *, int]
This converts a C string and its lengths to a Python object. If the C string pointer is NULL, None is returned.

z (str or None) [const char *]
Same as s.

z# (str or None) [const char *, int]
Same as s#.

u (str) [const wchar_t *]
Convert a null-terminated wchar_t buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

u# (str) [const wchar_t *, int]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

U (str or None) [const char *]
Same as \texttt{s}.

\textbf{U# (str or None) [const char *, int]}

Same as \texttt{s#}.

\textbf{i (int) [int]}

Convert a plain C \texttt{int} to a Python integer object.

\textbf{b (int) [char]}

Convert a plain C \texttt{char} to a Python integer object.

\textbf{h (int) [short int]}

Convert a plain C \texttt{short int} to a Python integer object.

\textbf{l (int) [long int]}

Convert a C \texttt{long int} to a Python integer object.

\textbf{B (int) [unsigned char]}

Convert a C unsigned \texttt{char} to a Python integer object.

\textbf{H (int) [unsigned short int]}

Convert a C unsigned \texttt{short int} to a Python integer object.

\textbf{I (int) [unsigned int]}

Convert a C unsigned \texttt{int} to a Python integer object.

\textbf{k (int) [unsigned long]}

Convert a C unsigned \texttt{long} to a Python integer object.

\textbf{L (int) [long long]}

Convert a C \texttt{long long} to a Python integer object.

\textbf{K (int) [unsigned long long]}

Convert a C unsigned \texttt{long long} to a Python integer object.

\textbf{n (int) [Py_ssize_t]}


Convert a C Py_ssize_t to a Python integer.

c (bytes of length 1) [char]
Convert a C int representing a byte to a Python bytes object of length 1.

C (str of length 1) [int]
Convert a C int representing a character to Python str object of length 1.

d (float) [double]
Convert a C double to a Python floating point number.

f (float) [float]
Convert a C float to a Python floating point number.

D (complex) [Py_complex *]
Convert a C Py_complex structure to a Python complex number.

O (object) [PyObject *]
Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a NULL pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, Py_BuildValue() will return NULL but won’t raise an exception. If no exception has been raised yet, SystemError is set.

S (object) [PyObject *]
Same as O.

N (object) [PyObject *]
Same as O, except it doesn’t increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.
0& (object) [converter, anything]

Convert anything to a Python object through a converter function. The function is called with anything (which should be compatible with void *) as its argument and should return a “new” Python object, or NULL if an error occurred.

(items) (tuple) [matching-items]

Convert a sequence of C values to a Python tuple with the same number of items.

[items] (list) [matching-items]

Convert a sequence of C values to a Python list with the same number of items.

{items} (dict) [matching-items]

Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the SystemError exception is set and NULL returned.

PyObject* Py_VaBuildValue(const char *format, va_list vargs)


Identical to Py_BuildValue(), except that it accepts a va_list rather than a variable number of arguments.
String conversion and formatting

Functions for number conversion and formatted string output.

int PyOS_snprintf(char *str, size_t size, const char *format, ...)
    Output not more than size bytes to str according to the format string format and the extra arguments. See the Unix man page snprintf(2).

int PyOS_vsnprintf(char *str, size_t size, const char *format, va_list va)
    Output not more than size bytes to str according to the format string format and the variable argument list va. Unix man page vsnprintf(2).

PyOS_snprintf() and PyOS_vsnprintf() wrap the Standard C library functions snprintf() and vsnprintf(). Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that str[\*size-1] is always '\0' upon return. They never write more than size bytes (including the trailing '\0') into str. Both functions require that str != NULL, size > 0 and format != NULL.

If the platform doesn't have vsnprintf() and the buffer size needed to avoid truncation exceeds size by more than 512 bytes, Python aborts with a Py_FatalError.

The return value (rv) for these functions should be interpreted as follows:
• When $0 \leq rv < \text{size}$, the output conversion was successful and $rv$ characters were written to $str$ (excluding the trailing '0' byte at $str[rv]$).
• When $rv \geq \text{size}$, the output conversion was truncated and a buffer with $rv + 1$ bytes would have been needed to succeed. $str[\text{size}-1]$ is '\0' in this case.
• When $rv < 0$, “something bad happened.” $str[\text{size}-1]$ is '\0' in this case too, but the rest of $str$ is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

```c
double PyOS_string_to_double(const char *s, char **endptr, PyObject *overflow_exception)
```

Convert a string $s$ to a double, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's `float()` constructor, except that $s$ must not have leading or trailing whitespace. The conversion is independent of the current locale.

If $\text{endptr}$ is NULL, convert the whole string. Raise `ValueError` and return -1.0 if the string is not a valid representation of a floating-point number.

If $\text{endptr}$ is not NULL, convert as much of the string as possible and set $\text{endptr}$ to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set $\text{endptr}$ to point to the beginning of the string, raise `ValueError`, and return -1.0.

If $s$ represents a value that is too large to store in a float (for example, "1e500" is such a string on many platforms) then if
overflow_exception is NULL return Py_HUGE_VAL (with an appropriate sign) and don’t set any exception. Otherwise, overflow_exception must point to a Python exception object; raise that exception and return -1.0. In both cases, set *endptr to point to the first character after the converted value.

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return -1.0.

New in version 3.1.

char* PyOS_double_to_string(double val, char format_code, int precision, int flags, int *ptype)

Convert a double val to a string using supplied format_code, precision, and flags.

format_code must be one of 'e', 'E', 'f', 'F', 'g', 'G' or 'r'. For 'r', the supplied precision must be 0 and is ignored. The 'r' format code specifies the standard repr() format.

flags can be zero or more of the values Py_DTSF_SIGN, Py_DTSF_ADD_DOT_0, or Py_DTSF_ALT, or-ed together:

- Py_DTSF_SIGN means to always precede the returned string with a sign character, even if val is non-negative.
- Py_DTSF_ADD_DOT_0 means to ensure that the returned string will not look like an integer.
- Py_DTSF_ALT means to apply “alternate” formatting rules. See the documentation for the PyOS_snprintf() '#' specifier for details.

If ptype is non-NULL, then the value it points to will be set to one of Py_DTST_FINITE, Py_DTST_INFINITE, or Py_DTST_NAN,
signifying that \textit{val} is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to \textit{buffer} with the converted string or \textit{NULL} if the conversion failed. The caller is responsible for freeing the returned string by calling \texttt{PyMem_Free()}. 

\textit{New in version 3.1.}

\begin{verbatim}
int PyOS_stricmp(const char *s1, const char *s2)
    Case insensitive comparison of strings. The function works almost identically to \texttt{strcmp()} except that it ignores the case.

int PyOS_strnicmp(const char *s1, const char *s2, Py_ssize_t size)
    Case insensitive comparison of strings. The function works almost identically to \texttt{strncmp()} except that it ignores the case.
\end{verbatim}
Reflection

PyObject* PyEval_GetBuiltins()

*Return value: Borrowed reference.*
Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

PyObject* PyEval_GetLocals()

*Return value: Borrowed reference.*
Return a dictionary of the local variables in the current execution frame, or NULL if no frame is currently executing.

PyObject* PyEval_GetGlobals()

*Return value: Borrowed reference.*
Return a dictionary of the global variables in the current execution frame, or NULL if no frame is currently executing.

PyFrameObject* PyEval_GetFrame()

*Return value: Borrowed reference.*
Return the current thread state's frame, which is NULL if no frame is currently executing.

int PyFrame_GetLineNumber(PyFrameObject *frame)

Return the line number that frame is currently executing.

const char* PyEval_GetFuncName(PyObject *func)

Return the name of func if it is a function, class or instance object, else the name of funcs type.

const char* PyEval_GetFuncDesc(PyObject *func)

Return a description string, depending on the type of func. Return values include “()” for functions and methods, ” constructor”, ”
instance”, and " object”. Concatenated with the result of PyEval_GetFuncName(), the result will be a description of func.
Codec registry and support functions

int **PyCodec_Register**(PyObject *search_function)
Register a new codec search function.

As side effect, this tries to load the encodings package, if not yet done, to make sure that it is always first in the list of search functions.

int **PyCodec_KnownEncoding**(const char *encoding)
Return 1 or 0 depending on whether there is a registered codec for the given encoding. This function always succeeds.

PyObject** PyCodec_Encode**(PyObject *object, const char *encoding, const char *errors)
*Return value: New reference.*
Generic codec based encoding API.

*object* is passed through the encoder function found for the given encoding using the error handling method defined by *errors*. *errors* may be NULL to use the default method defined for the codec. Raises a LookupError if no encoder can be found.

PyObject** PyCodec_Decode**(PyObject *object, const char *encoding, const char *errors)
*Return value: New reference.*
Generic codec based decoding API.

*object* is passed through the decoder function found for the given encoding using the error handling method defined by *errors*. *errors*
may be *NULL* to use the default method defined for the codec. Raises a *LookupError* if no encoder can be found.
Codec lookup API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a *KeyError* is set and *NULL* returned.

**PyObject** PyCodec_Encoder(const char *encoding)

*Return value: New reference.*

Get an encoder function for the given *encoding*.

**PyObject** PyCodec_Decoder(const char *encoding)

*Return value: New reference.*

Get a decoder function for the given *encoding*.

**PyObject** PyCodec_IncrementalEncoder(const char *encoding, const char *errors)

*Return value: New reference.*

Get an IncrementalEncoder object for the given *encoding*.

**PyObject** PyCodec_IncrementalDecoder(const char *encoding, const char *errors)

*Return value: New reference.*

Get an IncrementalDecoder object for the given *encoding*.

**PyObject** PyCodec_StreamReader(const char *encoding, PyObject *stream, const char *errors)

*Return value: New reference.*

Get a StreamReader factory function for the given *encoding*.

**PyObject** PyCodec_StreamWriter(const char *encoding, PyObject *stream, const char *errors)
Get a StreamWriter factory function for the given encoding.
Registry API for Unicode encoding error handlers

```c
int PyCodec_RegisterError(const char *name,
PyObject *error)
```

Register the error handling callback function `error` under the given `name`. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and `name` is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see Unicode Exception Objects for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

```c
PyObject* PyCodec.LookupError(const char *name)
```

Lookup the error handling callback function registered under `name`. As a special case `NULL` can be passed, in which case the error handling callback for “strict” will be returned.

```c
PyObject* PyCodec.StrictErrors(PyObject *exc)
```

Return value: Always `NULL`. 
Raise exc as an exception.

```
PyObject* PyCodec_IgnoreErrors(PyObject *exc)
    Ignore the unicode error, skipping the faulty input.
```

```
PyObject* PyCodec_ReplaceErrors(PyObject *exc)
    Replace the unicode encode error with ? or U+FFFD.
```

```
PyObject*
PyCodec_XMLCharRefReplaceErrors(PyObject *exc)
    Replace the unicode encode error with XML character references.
```

```
PyObject*
PyCodec_BackslashReplaceErrors(PyObject *exc)
    Replace the unicode encode error with backslash escapes (\\x, \u and \U).
```

```
PyObject* PyCodec_NameReplaceErrors(PyObject *exc)
    Replace the unicode encode error with \N{...} escapes.
```

New in version 3.5.
Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

It is not possible to use these functions on objects that are not properly initialized, such as a list object that has been created by `PyList_New()`, but whose items have not been set to some non-NULL value yet.

- Object Protocol
- Number Protocol
- Sequence Protocol
- Mapping Protocol
- Iterator Protocol
- Buffer Protocol
  - Buffer structure
  - Buffer request types
    - request-independent fields
    - readonly, format
    - shape, strides, suboffsets
    - contiguity requests
    - compound requests
  - Complex arrays
    - NumPy-style: shape and strides
    - PIL-style: shape, strides and suboffsets
  - Buffer-related functions
- Old Buffer Protocol
Object Protocol

PyObject* Py_NotImplemented

The NotImplemented singleton, used to signal that an operation is not implemented for the given type combination.

Py_RETURN_NOTIMPLEMENTED

Properly handle returning Py_NotImplemented from within a C function (that is, increment the reference count of NotImplemented and return it).

int PyObject_Print(PyObject *o, FILE *fp, int flags)

Print an object o, on file fp. Returns -1 on error. The flags argument is used to enable certain printing options. The only option currently supported is Py_PRINT_RAW; if given, the str() of the object is written instead of the repr().

int PyObject_HasAttr(PyObject *o, PyObject *attr_name)

Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression hasattr(o, attr_name). This function always succeeds.

Note that exceptions which occur while calling __getattr__() and __getattribute__() methods will get suppressed. To get error reporting use PyObject_GetAttr() instead.

int PyObject_HasAttrString(PyObject *o, const char *attr_name)

Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression hasattr(o, attr_name). This function always succeeds.
Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods and creating a temporary string object will get suppressed. To get error reporting use `PyObject_GetAttrString()` instead.

```c
PyObject* PyObject_GetAttr(PyObject *o, PyObject *attr_name)

Retrieve an attribute named `attr_name` from object `o`. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

PyObject* PyObject_GetAttrString(PyObject *o, const char *attr_name)

Retrieve an attribute named `attr_name` from object `o`. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

PyObject* PyObject_GenericGetAttr(PyObject *o, PyObject *name)

Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in Implementing Descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int PyObject_SetAttr(PyObject *o, PyObject *attr_name, PyObject *v)
```
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement *o.attr_name = v*.

If *v* is NULL, the attribute is deleted, however this feature is deprecated in favour of using PyObject_DelAttr().

```c
int PyObject_SetAttrString(PyObject *o, const char *attr_name, PyObject *v) {
    Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement *o.attr_name = v*.

    If *v* is NULL, the attribute is deleted, however this feature is deprecated in favour of using PyObject_DelAttrString().
}
```

```c
int PyObject_GenericSetAttr(PyObject *o, PyObject *name, PyObject *value) {
    Generic attribute setter and deleter function that is meant to be put into a type object's *tp_setattro* slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's *__dict__* (if present). On success, 0 is returned, otherwise an AttributeError is raised and -1 is returned.
}
```

```c
int PyObject_DelAttr(PyObject *o, PyObject *attr_name) {
    Delete attribute named *attr_name*, for object *o*. Returns -1 on failure. This is the equivalent of the Python statement *del o.attr_name*.
}
```
int **PyObject** DelAttrString**(PyObject** *o, const char *attr_name)**
    Delete attribute named *attr_name*, for object *o*. Returns -1 on failure. This is the equivalent of the Python statement `del o.attr_name`.

**PyObject** *PyObject** GenericGetDict**(PyObject** *o, void *context)**
    *Return value: New reference.*
    A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.
    
    *New in version 3.3.*

int **PyObject** GenericSetDict**(PyObject** *o, void *context)**
    A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.
    
    *New in version 3.3.*

**PyObject** *PyObject** RichCompare**(PyObject** *o1, PyObject *o2, int opid)**
    *Return value: New reference.*
    Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, ==, !=, >, or >= respectively. This is the equivalent of the Python expression `o1 op o2`, where *op* is the operator corresponding to *opid*. Returns the value of the comparison on success, or NULL on failure.

int **PyObject** RichCompareBool**(PyObject** *o1, PyObject *o2, int opid)**
Compare the values of o1 and o2 using the operation specified by opid, which must be one of Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, ==, !=, >, or >= respectively. Returns -1 on error, 0 if the result is false, 1 otherwise. This is the equivalent of the Python expression o1 op o2, where op is the operator corresponding to opid.

**Note:** If o1 and o2 are the same object, `PyObject_RichCompareBool()` will always return 1 for Py_EQ and 0 for Py_NE.

**PyObject**\* **PyObject_Repr**(PyObject \*o)

*Return value: New reference.*

Compute a string representation of object o. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

*Changed in version 3.4:* This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

**PyObject**\* **PyObject_ASCII**(PyObject \*o)

*Return value: New reference.*

As `PyObject_Repr()`, compute a string representation of object o, but escape the non-ASCII characters in the string returned by `PyObject_Repr()` with \x, \u or \U escapes. This generates a string similar to that returned by `PyObject_Repr()` in Python 2. Called by the `ascii()` built-in function.

**PyObject**\* **PyObject_Str**(PyObject \*o)

*Return value: New reference.*
Compute a string representation of object o. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression str(o). Called by the str() built-in function and, therefore, by the print() function.

*Changed in version 3.4:* This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

**PyObject**\*`PyObject_Bytes(PyObject \*o)`

*Return value: New reference.*

Compute a bytes representation of object o. NULL is returned on failure and a bytes object on success. This is equivalent to the Python expression bytes(o), when o is not an integer. Unlike bytes(o), a TypeError is raised when o is an integer instead of a zero-initialized bytes object.

**int**`PyObject_IsSubclass(PyObject \*derived, PyObject \*cls)`

Return 1 if the class derived is identical to or derived from the class cls, otherwise return 0. In case of an error, return -1.

If cls is a tuple, the check will be done against every entry in cls. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0.

If cls has a `__subclasscheck__` method, it will be called to determine the subclass status as described in PEP 3119. Otherwise, derived is a subclass of cls if it is a direct or indirect subclass, i.e. contained in cls.__mro__.

Normally only class objects, i.e. instances of type or a derived class, are considered classes. However, objects can override this
by having a __bases__ attribute (which must be a tuple of base classes).

int PyObject_IsInstance(PyObject *inst, PyObject *cls)
Return 1 if inst is an instance of the class cls or a subclass of cls, or 0 if not. On error, returns -1 and sets an exception.

If cls is a tuple, the check will be done against every entry in cls. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0.

If cls has a __instancecheck__() method, it will be called to determine the subclass status as described in PEP 3119. Otherwise, inst is an instance of cls if its class is a subclass of cls.

An instance inst can override what is considered its class by having a __class__ attribute.

An object cls can override if it is considered a class, and what its base classes are, by having a __bases__ attribute (which must be a tuple of base classes).

int PyCallable_Check(PyObject *o)
Determine if the object o is callable. Return 1 if the object is callable and 0 otherwise. This function always succeeds.

PyObject* PyObject_Call(PyObject *callable, PyObject *args, PyObject *kwargs)

Call a callable Python object callable, with arguments given by the tuple args, and named arguments given by the dictionary kwargs.

args must not be NULL, use an empty tuple if no arguments are
needed. If no named arguments are needed, \textit{kwargs} can be \textit{NULL}.

Returns the result of the call on success, or \textit{NULL} on failure.

This is the equivalent of the Python expression: \texttt{callable(*args, **kwargs)}.

\textbf{PyObject* \texttt{PyObject\_CallObject}(PyObject *callable, PyObject *args)}

\textit{Return value: New reference.}
Call a callable Python object \texttt{callable}, with arguments given by the tuple \texttt{args}. If no arguments are needed, then \texttt{args} can be \textit{NULL}.

Returns the result of the call on success, or \textit{NULL} on failure.

This is the equivalent of the Python expression: \texttt{callable(*args)}.

\textbf{PyObject* \texttt{PyObject\_CallFunction}(PyObject *callable, const char *format, ...)}

\textit{Return value: New reference.}
Call a callable Python object \texttt{callable}, with a variable number of C arguments. The C arguments are described using a \texttt{Py\_BuildValue()} style format string. The format can be \textit{NULL}, indicating that no arguments are provided.

Returns the result of the call on success, or \textit{NULL} on failure.

This is the equivalent of the Python expression: \texttt{callable(*args)}.

Note that if you only pass \texttt{PyObject * args}, \texttt{PyObject\_CallFunctionObjArgs()} is a faster alternative.
**Changed in version 3.4:** The type of `format` was changed from `char *`.

```c
PyObject* PyObject_CallMethod(PyObject *obj, const char *name, const char *format, ...)
```

*Return value: New reference.*

Call the method named `name` of object `obj` with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple.

The format can be `NULL`, indicating that no arguments are provided.

Returns the result of the call on success, or `NULL` on failure.

This is the equivalent of the Python expression: `obj.name(arg1, arg2, ...)`.  

Note that if you only pass `PyObject *` `args`, `PyObject_CallMethodObjArgs()` is a faster alternative.

**Changed in version 3.4:** The types of `name` and `format` were changed from `char *`.

```c
PyObject*
PyObject_CallFunctionObjArgs(PyObject *callable, ..., NULL)
```

*Return value: New reference.*

Call a callable Python object `callable`, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by `NULL`.

Returns the result of the call on success, or `NULL` on failure.
This is the equivalent of the Python expression: `callable(arg1, arg2, ...)`.  

```c
PyObject* PyObject_CallMethodObjArgs(PyObject *obj, PyObject *name, ..., NULL)
```

*Return value: New reference.*

Calls a method of the Python object `obj`, where the name of the method is given as a Python string object in `name`. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by `NULL`. Returns the result of the call on success, or `NULL` on failure.

```c
Py_hash_t PyObject_Hash(PyObject *o)
```

Compute and return the hash value of an object `o`. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

*Changed in version 3.2:* The return type is now `Py_hash_t`. This is a signed integer the same size as `Py_ssize_t`.

```c
Py_hash_t PyObject_HashNotImplemented(PyObject *o)
```

Set a `TypeError` indicating that `type(o)` is not hashable and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

```c
int PyObject_IsTrue(PyObject *o)
```

Returns `1` if the object `o` is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

```c
int PyObject_Not(PyObject *o)
```
Returns 0 if the object o is considered to be true, and 1 otherwise. This is equivalent to the Python expression `not o`. On failure, return -1.

**PyObject** PyObject_Type(PyObject *o)

*Return value: New reference.*

When o is non-NULL, returns a type object corresponding to the object type of object o. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression `o->ob_type`, which returns a pointer of type `PyTypeObject *`, except when the incremented reference count is needed.

**int** PyObject_TypeCheck(PyObject *o, PyTypeObject *type)

Return true if the object o is of type type or a subtype of type. Both parameters must be non-NULL.

**Py_ssize_t** PyObject_Size(PyObject *o)

**Py_ssize_t** PyObject_Length(PyObject *o)

Return the length of object o. If the object o provides either the sequence and mapping protocols, the sequence length is returned. On error, -1 is returned. This is the equivalent to the Python expression `len(o)`. On failure, return -1.

**Py_ssize_t** PyObject_LengthHint(PyObject *o, Py_ssize_t default)

Return an estimated length for the object o. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return -1. This is the equivalent to the Python expression...
operator.length_hint(o, default).

*New in version 3.4.*

**PyObject* PyObject_GetItem*(PyObject *o, PyObject *key)*

*Return value: New reference.*

Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression o[key].

**int** PyObject_SetItem*(PyObject *o, PyObject *key, PyObject *v)*

Map the object key to the value v. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement o[key] = v.

**int** PyObject_DelItem*(PyObject *o, PyObject *key)*

Remove the mapping for the object key from the object o. Return -1 on failure. This is equivalent to the Python statement del o[key].

**PyObject** PyObject_Dir*(PyObject *o)*

*Return value: New reference.*

This is equivalent to the Python expression dir(o), returning a (possibly empty) list of strings appropriate for the object argument, or NULL if there was an error. If the argument is NULL, this is like the Python dir(), returning the names of the current locals; in this case, if no execution frame is active then NULL is returned but PyErr_Occurred() will return false.

**PyObject** PyObject_GetIter*(PyObject *o)*

*Return value: New reference.*

This is equivalent to the Python expression iter(o). It returns a new iterator for the object argument, or the object itself if the object
is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.
Number Protocol

int PyNumber_Check(PyObject *o)
    Returns 1 if the object o provides numeric protocols, and false otherwise. This function always succeeds.

PyObject* PyNumber_Add(PyObject *o1, PyObject *o2)
    Returns the result of adding o1 and o2, or NULL on failure. This is the equivalent of the Python expression o1 + o2.

PyObject* PyNumber_Subtract(PyObject *o1, PyObject *o2)
    Returns the result of subtracting o2 from o1, or NULL on failure. This is the equivalent of the Python expression o1 - o2.

PyObject* PyNumber_Multiply(PyObject *o1, PyObject *o2)
    Returns the result of multiplying o1 and o2, or NULL on failure. This is the equivalent of the Python expression o1 * o2.

PyObject* PyNumber_MatrixMultiply(PyObject *o1, PyObject *o2)
    Returns the result of matrix multiplication on o1 and o2, or NULL on failure. This is the equivalent of the Python expression o1 @ o2.

    New in version 3.5.

PyObject* PyNumber_FloorDivide(PyObject *o1,
PyObject *o2)

Return the floor of o1 divided by o2, or NULL on failure. This is equivalent to the “classic” division of integers.

PyObject* PyNumber_TrueDivide(PyObject *o1, PyObject *o2)

Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers.

PyObject* PyNumber_Remainder(PyObject *o1, PyObject *o2)

Returns the remainder of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression o1 % o2.

PyObject* PyNumber_Divmod(PyObject *o1, PyObject *o2)

See the built-in function divmod(). Returns NULL on failure. This is the equivalent of the Python expression divmod(o1, o2).

PyObject* PyNumber_Power(PyObject *o1, PyObject *o2, PyObject *o3)

See the built-in function pow(). Returns NULL on failure. This is the equivalent of the Python expression pow(o1, o2, o3), where o3 is optional. If o3 is to be ignored, pass Py_None in its place (passing NULL for o3 would cause an illegal memory access).
PyObject* **PyNumber_Negative**(PyObject *o)

*Return value: New reference.*
Returns the negation of o on success, or *NULL* on failure. This is the equivalent of the Python expression `-o`.

PyObject* **PyNumber_Positive**(PyObject *o)

*Return value: New reference.*
Returns o on success, or *NULL* on failure. This is the equivalent of the Python expression `+o`.

PyObject* **PyNumber_Absolute**(PyObject *o)

*Return value: New reference.*
Returns the absolute value of o, or *NULL* on failure. This is the equivalent of the Python expression `abs(o)`.

PyObject* **PyNumber_Invert**(PyObject *o)

*Return value: New reference.*
Returns the bitwise negation of o on success, or *NULL* on failure. This is the equivalent of the Python expression `~o`.

PyObject* **PyNumber_Lshift**(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the result of left shifting o1 by o2 on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 << o2`.

PyObject* **PyNumber_Rshift**(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the result of right shifting o1 by o2 on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 >> o2`.

PyObject* **PyNumber_And**(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the “bitwise and” of o1 and o2 on success and *NULL* on failure.
failure. This is the equivalent of the Python expression $o1 \& o2$.

**PyObject** PyNumber_Xor**(PyObject *o1, PyObject *o2)**  
*Return value: New reference.*  
Returns the “bitwise exclusive or” of $o1$ by $o2$ on success, or NULL on failure. This is the equivalent of the Python expression $o1 \^ o2$.

**PyObject** PyNumber_Or**(PyObject *o1, PyObject *o2)**  
*Return value: New reference.*  
Returns the “bitwise or” of $o1$ and $o2$ on success, or NULL on failure. This is the equivalent of the Python expression $o1 \mid o2$.

**PyObject** PyNumber_InPlaceAdd**(PyObject *o1, PyObject *o2)**  
*Return value: New reference.*  
Returns the result of adding $o1$ and $o2$, or NULL on failure. The operation is done *in-place* when $o1$ supports it. This is the equivalent of the Python statement $o1 += o2$.

**PyObject** PyNumber_InPlaceSubtract**(PyObject *o1, PyObject *o2)**  
*Return value: New reference.*  
Returns the result of subtracting $o2$ from $o1$, or NULL on failure. The operation is done *in-place* when $o1$ supports it. This is the equivalent of the Python statement $o1 -= o2$.

**PyObject** PyNumber_InPlaceMultiply**(PyObject *o1, PyObject *o2)**  
*Return value: New reference.*  
Returns the result of multiplying $o1$ and $o2$, or NULL on failure. The operation is done *in-place* when $o1$ supports it. This is the equivalent of the Python statement $o1 *= o2$. 

PyObject*
PyNumber_InPlaceMatrixMultiply(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the result of matrix multiplication on o1 and o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 \@= o2.

*New in version 3.5.*

PyObject*
PyNumber_InPlaceFloorDivide(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the mathematical floor of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 //\= o2.

PyObject*
PyNumber_InPlaceTrueDivide(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when o1 supports it.

PyObject*
PyNumber_InPlaceRemainder(PyObject *o1, PyObject *o2)

*Return value: New reference.*
Returns the remainder of dividing o1 by o2, or NULL on failure. The operation is done *in-place* when o1 supports it. This is the equivalent of the Python statement o1 \%= o2.
PyObject* PyNumber_InPlacePower(PyObject *o1, PyObject *o2, PyObject *o3)

See the built-in function pow(). Returns NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 **= o2 when o3 is Py_None, or an in-place variant of pow(o1, o2, o3) otherwise. If o3 is to be ignored, pass Py_None in its place (passing NULL for o3 would cause an illegal memory access).

PyObject* PyNumber_InPlaceLshift(PyObject *o1, PyObject *o2)

Returns the result of left shifting o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 <<= o2.

PyObject* PyNumber_InPlaceRshift(PyObject *o1, PyObject *o2)

Returns the result of right shifting o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 >>= o2.

PyObject* PyNumber_InPlaceAnd(PyObject *o1, PyObject *o2)

Returns the “bitwise and” of o1 and o2 on success and NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 &= o2.

PyObject* PyNumber_InPlaceXor(PyObject *o1, PyObject *o2)

Returns the “bitwise exclusive or” of o1 by o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 ^= o2.

PyObject* PyNumber_InPlaceOr(PyObject *o1, PyObject *o2)

Returns the “bitwise or” of o1 and o2 on success, or NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python statement o1 |= o2.

PyObject* PyNumber_Long(PyObject *o)

Returns the o converted to an integer object on success, or NULL on failure. This is the equivalent of the Python expression int(o).

PyObject* PyNumber_Float(PyObject *o)

Returns the o converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression float(o).

PyObject* PyNumber_Index(PyObject *o)

Returns the o converted to a Python int on success or NULL with a TypeError exception raised on failure.

PyObject* PyNumber_ToBase(PyObject *n, int base)

Returns the integer n converted to base base as a string. The base argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of '0b', '0o', or '0x', respectively. If n is not a Python int, it is converted with PyNumber_Index() first.
Pyssize_t PyNumber_AsSsize_t(PyObject *o, PyObject *exc)

Returns o converted to a Pyssize_t value if o can be interpreted as an integer. If the call fails, an exception is raised and -1 is returned.

If o can be converted to a Python int but the attempt to convert to a Pyssize_t value would raise an OverflowError, then the exc argument is the type of exception that will be raised (usually IndexError or OverflowError). If exc is NULL, then the exception is cleared and the value is clipped to PY_SSIZE_T_MIN for a negative integer or PY_SSIZE_T_MAX for a positive integer.

int PyIndex_Check(PyObject *o)

Returns 1 if o is an index integer (has the nb_index slot of the tp_as_number structure filled in), and 0 otherwise. This function always succeeds.
Sequence Protocol

int PySequence_Check(PyObject *o)
    Return 1 if the object provides sequence protocol, and 0 otherwise. Note that it returns 1 for Python classes with a __getitem__() method unless they are dict subclasses since in general case it is impossible to determine what the type of keys it supports. This function always succeeds.

Py_ssize_t PySequence_Size(PyObject *o)
Py_ssize_t PySequence_Length(PyObject *o)
    Returns the number of objects in sequence o on success, and -1 on failure. This is equivalent to the Python expression len(o).

PyObject* PySequence_Concat(PyObject *o1, PyObject *o2)
    Return the concatenation of o1 and o2 on success, and NULL on failure. This is the equivalent of the Python expression o1 + o2.

PyObject* PySequence_Repeat(PyObject *o, Py_ssize_t count)
    Return the result of repeating sequence object o count times, or NULL on failure. This is the equivalent of the Python expression o * count.

PyObject* PySequence_InPlaceConcat(PyObject *o1, PyObject *o2)
    Return the concatenation of o1 and o2 on success, and NULL on failure. The operation is done in-place when o1 supports it. This is the equivalent of the Python expression o1 += o2.
PyObject* PySequence_InPlaceRepeat(PyObject *o, Py_ssize_t count)

Return the result of repeating sequence object o count times, or NULL on failure. The operation is done in-place when o supports it. This is the equivalent of the Python expression o *= count.

PyObject* PySequence_GetItem(PyObject *o, Py_ssize_t i)

Return the i-th element of o, or NULL on failure. This is the equivalent of the Python expression o[i].

PyObject* PySequence_GetSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2)

Return the slice of sequence object o between i1 and i2, or NULL on failure. This is the equivalent of the Python expression o[i1:i2].

int PySequence_SetItem(PyObject *o, Py_ssize_t i, PyObject *v)

Assign object v to the i-th element of o. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement o[i] = v. This function does not steal a reference to v.

If v is NULL, the element is deleted, however this feature is deprecated in favour of using PySequence_DelItem().

int PySequence_DelItem(PyObject *o, Py_ssize_t i)

Delete the i-th element of object o. Returns -1 on failure. This is the equivalent of the Python statement del o[i].
int PySequence_SetSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2, PyObject *v)
    Assign the sequence object v to the slice in sequence object o from i1 to i2. This is the equivalent of the Python statement o[i1:i2] = v.

int PySequence_DelSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2)
    Delete the slice in sequence object o from i1 to i2. Returns -1 on failure. This is the equivalent of the Python statement del o[i1:i2].

Py_ssize_t PySequence_Count(PyObject *o, PyObject *value)
    Return the number of occurrences of value in o, that is, return the number of keys for which o[key] == value. On failure, return -1. This is equivalent to the Python expression o.count(value).

int PySequence_Contains(PyObject *o, PyObject *value)
    Determine if o contains value. If an item in o is equal to value, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression value in o.

Py_ssize_t PySequence_Index(PyObject *o, PyObject *value)
    Return the first index i for which o[i] == value. On error, return -1. This is equivalent to the Python expression o.index(value).

PyObject* PySequence_List(PyObject *o)
    *Return value: New reference.*
    Return a list object with the same contents as the sequence or iterable o, or NULL on failure. The returned list is guaranteed to be
new. This is equivalent to the Python expression `list(o)`.

```
PyObject* PySequence_Tuple(PyObject *o)


Return a tuple object with the same contents as the sequence or iterable `o`, or `NULL` on failure. If `o` is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.
```

```
PyObject* PySequence_Fast(PyObject *o, const char *m)


Return the sequence or iterable `o` as a list, unless it is already a tuple or list, in which case `o` is returned. Use `PySequence_Fast_GET_ITEM()` to access the members of the result. Returns `NULL` on failure. If the object is not a sequence or iterable, raises `TypeError` with `m` as the message text.
```

```
Py_ssize_t PySequence_Fast_GET_SIZE(PyObject *o)

Returns the length of `o`, assuming that `o` was returned by `PySequence_Fast()` and that `o` is not `NULL`. The size can also be gotten by calling `PySequence_Size()` on `o`, but `PySequence_Fast_GET_SIZE()` is faster because it can assume `o` is a list or tuple.
```

```
PyObject* PySequence_Fast_GET_ITEM(PyObject *o, Py_ssize_t i)

Return value: Borrowed reference.

Return the `i`th element of `o`, assuming that `o` was returned by `PySequence_Fast()`, `o` is not `NULL`, and that `i` is within bounds.
```

```
PyObject** PySequence_Fast_ITEMS(PyObject *o)

Return the underlying array of PyObject pointers. Assumes that `o`
was returned by `PySequence_Fast()` and `o` is not `NULL`.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

**PyObject** `PySequence_ITEM(PyObject *o, Py_ssize_t i)`

*Return value: New reference.*

Return the `ith` element of `o` or `NULL` on failure. Macro form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on `o` is true and without adjustment for negative indices.
Mapping Protocol

See also `PyObject_GetItem()`, `PyObject_SetItem()` and `PyObject_DelItem()`.

```c
int PyMapping_Check(PyObject *o)
```

Return 1 if the object provides mapping protocol or supports slicing, and 0 otherwise. Note that it returns 1 for Python classes with a `__getitem__()` method since in general case it is impossible to determine what the type of keys it supports. This function always succeeds.

```c
Py_ssize_t PyMapping_Size(PyObject *o)
Py_ssize_t PyMapping_Length(PyObject *o)
```

Returns the number of keys in object o on success, and -1 on failure. This is equivalent to the Python expression `len(o)`.

```c
PyObject* PyMapping_GetItemString(PyObject *o, const char *key)
```

*Return value: New reference.*

Return element of o corresponding to the string key or NULL on failure. This is the equivalent of the Python expression `o[key]`. See also `PyObject_GetItem()`.

```c
int PyMapping_SetItemString(PyObject *o, const char *key, PyObject *v)
```

Map the string key to the value v in object o. Returns -1 on failure. This is the equivalent of the Python statement `o[key] = v`. See also `PyObject_SetItem()`.

```c
int PyMapping_DelItem(PyObject *o, PyObject *key)
```
Remove the mapping for the object key from the object o. Return -1 on failure. This is equivalent to the Python statement del o[key]. This is an alias of PyObject_DelItem().

int PyMapping_DelItemString(PyObject *o, const char *key)  
Remove the mapping for the string key from the object o. Return -1 on failure. This is equivalent to the Python statement del o[key].

int PyMapping_HasKey(PyObject *o, PyObject *key)  
Return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to the Python expression key in o. This function always succeeds.

Note that exceptions which occur while calling the __getitem__() method will get suppressed. To get error reporting use PyObject_GetItem() instead.

int PyMapping_HasKeyString(PyObject *o, const char *key)  
Return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to the Python expression key in o. This function always succeeds.

Note that exceptions which occur while calling the __getitem__() method and creating a temporary string object will get suppressed. To get error reporting use PyMapping_GetItemString() instead.

PyObject* PyMapping_Keys(PyObject *o)  
On success, return a list of the keys in object o. On failure, return NULL.
Changed in version 3.7: Previously, the function returned a list or a tuple.

PyObject* PyMapping_Values(PyObject *o)

*Return value: New reference.*
On success, return a list of the values in object o. On failure, return NULL.

Changed in version 3.7: Previously, the function returned a list or a tuple.

PyObject* PyMapping_Items(PyObject *o)

*Return value: New reference.*
On success, return a list of the items in object o, where each item is a tuple containing a key-value pair. On failure, return NULL.

Changed in version 3.7: Previously, the function returned a list or a tuple.
Iterator Protocol

There are two functions specifically for working with iterators.

```c
int PyIter_Check(PyObject *o)
    Return true if the object o supports the iterator protocol.

PyObject* PyIter_Next(PyObject *o)
    *Return value: New reference.*
    Return the next value from the iteration o. The object must be an iterator (it is up to the caller to check this). If there are no remaining values, returns NULL with no exception set. If an error occurs while retrieving the item, returns NULL and passes along the exception.
```

To write a loop which iterates over an iterator, the C code should look something like this:

```c
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while (item = PyIter_Next(iterator)) {
    /* do something with item */
    ... /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
```
/* propagate error */
} else {
    /* continue doing useful work */
}
Buffer Protocol

Certain objects available in Python wrap access to an underlying memory array or buffer. Such objects include the built-in bytes and bytearray, and some extension types like array.array. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

Python provides such a facility at the C level in the form of the buffer protocol. This protocol has two sides:

- on the producer side, a type can export a “buffer interface” which allows objects of that type to expose information about their underlying buffer. This interface is described in the section Buffer Object Structures;
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter).

Simple objects such as bytes and bytearray expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an array.array can be multi-byte values.

An example consumer of the buffer interface is the write() method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While write() only needs
read-only access to the internal contents of the object passed to it, other methods such as \texttt{readinto()} need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call \texttt{PyObject\_GetBuffer()} with the right parameters;
- call \texttt{PyArg\_ParseTuple()} (or one of its siblings) with one of the \texttt{y*}, \texttt{w*} or \texttt{s*} format codes.

In both cases, \texttt{PyBuffer\_Release()} must be called when the buffer isn't needed anymore. Failure to do so could lead to various issues such as resource leaks.
Buffer structure

Buffer structures (or simply “buffers”) are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not *PyObject* pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a *memoryview* object can be created.

For short instructions how to write an exporting object, see Buffer Object Structures. For obtaining a buffer, see *PyObject_GetBuffer()*.

**Py_buffer**

```c
void *buf
```

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative *strides* the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.
void *obj

A new reference to the exporting object. The reference is owned by the consumer and automatically decremented and set to NULL by PyBuffer_Release(). The field is the equivalent of the return value of any standard C-API function.

As a special case, for temporary buffers that are wrapped by PyMemoryView_FromBuffer() or PyBuffer_FillInfo() this field is NULL. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t len

product(shape) * itemsize. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing ((char *)buf)[0] up to ((char *)buf)[len-1] is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be PyBUF_SIMPLE or PyBUF_WRITABLE.

int readonly

An indicator of whether the buffer is read-only. This field is controlled by the PyBUF_WRITABLE flag.

Py_ssize_t itemsize

Item size in bytes of a single element. Same as the value of struct.calcsize() called on non-NULL format values.

Important exception: If a consumer requests a buffer without the PyBUF_FORMAT flag, format will be set to NULL, but
itemsize still has the value for the original format.

If shape is present, the equality product(shape) * itemsize == len still holds and the consumer can use itemsize to navigate the buffer.

If shape is NULL as a result of a PyBUF_SIMPLE or a PyBUF_WRITABLE request, the consumer must disregard itemsize and assume itemsize == 1.

const char *format
A NULL terminated string in struct module style syntax describing the contents of a single item. If this is NULL, "B" (unsigned bytes) is assumed.

This field is controlled by the PyBUF_FORMAT flag.

int ndim
The number of dimensions the memory represents as an n-dimensional array. If it is 0, buf points to a single item representing a scalar. In this case, shape, strides and suboffsets MUST be NULL.

The macro PyBUF_MAX_NDIM limits the maximum number of dimensions to 64. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to PyBUF_MAX_NDIM dimensions.

Py_ssize_t *shape
An array of Py_ssize_t of length ndim indicating the shape of the memory as an n-dimensional array. Note that shape[0] * ... * shape[ndim-1] * itemsize MUST be equal to
Shape values are restricted to $\text{shape}[n] \geq 0$. The case $\text{shape}[n] == 0$ requires special attention. See complex arrays for further information.

The shape array is read-only for the consumer.

**Py_ssize_t**{*strides*}

An array of Py_ssize_t of length $\text{ndim}$ giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case $\text{strides}[n] \leq 0$. See complex arrays for further information.

The strides array is read-only for the consumer.

**Py_ssize_t**{*suboffsets*}

An array of Py_ssize_t of length $\text{ndim}$. If $\text{suboffsets}[n] \geq 0$, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be NULL (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See complex arrays for further information how to access elements of such an array.
The suboffsets array is read-only for the consumer.

void *internal

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.
Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via `PyObject_GetBuffer()`. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the `flags` argument to specify the exact buffer type it can handle.

All `Py_buffer` fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by `flags` and must always be filled in with the correct values: `obj, buf, len, itemsize, ndim`.

readonly, format

**PyBUF_WRITABLE**

Controls the `readonly` field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers.

**PyBUF_FORMAT**

Controls the `format` field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

`PyBUF_WRITABLE` can be |d to any of the flags in the next section. Since `PyBUF_SIMPLE` is defined as 0, `PyBUF_WRITABLE` can be used as a stand-alone flag to request a simple writable buffer.
PyBUF_FORMAT can be |'d to any of the flags except PyBUF_SIMPLE. The latter already implies format B (unsigned bytes).

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

<table>
<thead>
<tr>
<th>Request</th>
<th>shape</th>
<th>strides</th>
<th>suboffsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyBUF_INDIRECT</td>
<td>yes</td>
<td>yes</td>
<td>if needed</td>
</tr>
<tr>
<td>PyBUF_STRIDES</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
</tr>
<tr>
<td>PyBUF_ND</td>
<td>yes</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>PyBUF_SIMPLE</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

contiguity requests

C or Fortran contiguity can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

<table>
<thead>
<tr>
<th>Request</th>
<th>shape</th>
<th>strides</th>
<th>suboffsets</th>
<th>cont</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyBUF_C_CONTIGUOUS</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>C</td>
</tr>
<tr>
<td>PyBUF_F_CONTIGUOUS</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>F</td>
</tr>
<tr>
<td>PyBUF_ANY_CONTIGUOUS</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>C C</td>
</tr>
</tbody>
</table>
compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table $U$ stands for undefined contiguity. The consumer would have to call `PyBuffer_IsContiguous()` to determine contiguity.

<table>
<thead>
<tr>
<th>Request</th>
<th>shape</th>
<th>strides</th>
<th>suboffsets</th>
<th>contig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyBUF_FULL</td>
<td>yes</td>
<td>yes</td>
<td>if needed</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_FULL_RO</td>
<td>yes</td>
<td>yes</td>
<td>if needed</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_RECORDS</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_RECORDS_RO</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_STRIDED</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_STRIDED_RO</td>
<td>yes</td>
<td>yes</td>
<td>NULL</td>
<td>$U$</td>
</tr>
<tr>
<td>PyBUF_CONTIG</td>
<td>yes</td>
<td>NULL</td>
<td>NULL</td>
<td>$C$</td>
</tr>
<tr>
<td>PyBUF_CONTIG_RO</td>
<td>yes</td>
<td>NULL</td>
<td>NULL</td>
<td>$C$</td>
</tr>
</tbody>
</table>
Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by `itemsize`, `ndim`, `shape` and `strides`.

If `ndim == 0`, the memory location pointed to by `buf` is interpreted as a scalar of size `itemsize`. In that case, both `shape` and `strides` are `NULL`.

If `strides` is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```python
ptr = (char *)buf + indices[0] * strides[0] +
    ... + indices[n-1] * strides[n-1]
item = *((typeof(item) *)ptr);
```

As noted above, `buf` can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```python
def verify_structure(memlen, itemsize, ndim, shape, offset):
    """Verify that the parameters represent a valid the bounds of the allocated memory:
    char *mem: start of the physical memory block
    memlen: length of the physical memory block
    offset: (char *)buf - mem
    ""
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
```
if any(v % itemsize for v in strides):
    return False

if ndim <= 0:
    return ndim == 0 and not shape and not strides
if 0 in shape:
    return True

imin = sum(strides[j]*(shape[j]-1) for j in range(len(strides)) if strides[j] <= 0)
imax = sum(strides[j]*(shape[j]-1) for j in range(len(strides)) if strides[j] > 0)

return 0 <= offset+imin and offset+imax+itemsize

### PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of `buf`, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-NULL strides and suboffsets:

```c
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *suboffsets, Py_ssize_t itemsize) {
    char *pointer = (char*)buf;
```
int i;
for (i = 0; i < ndim; i++) {
    pointer += strides[i] * indices[i];
    if (suboffsets[i] >= 0) {
        pointer = *((char**)pointer) + suboffsets[i];
    }
}
return (void*)pointer;
Buffer-related functions

int **PyObject_CheckBuffer** *(PyObject *obj)*

Return 1 if *obj* supports the buffer interface otherwise 0. When 1 is returned, it doesn’t guarantee that **PyObject_GetBuffer()** will succeed. This function always succeeds.

int **PyObject_GetBuffer** *(PyObject *exporter, Py_buffer *view, int flags)*

Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise PyExc_BufferError, set view->obj to NULL and return -1.

On success, fill in *view*, set view->obj to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, view->obj MAY refer to this object instead of *exporter* (See Buffer Object Structures).

Successful calls to **PyObject_GetBuffer()** must be paired with calls to **PyBuffer_Release()**, similar to malloc() and free(). Thus, after the consumer is done with the buffer, **PyBuffer_Release()** must be called exactly once.

void **PyBuffer_Release** *(Py_buffer *view)*

Release the buffer *view* and decrement the reference count for view->obj. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via **PyObject_GetBuffer()**.
Py_ssize_t PyBuffer_SizeFromFormat(const char *)
    Return the implied itemsize from format. This function is not yet implemented.

int PyBuffer_IsContiguous(Py_buffer *view, char order)
    Return 1 if the memory defined by the view is C-style (order is 'C') or Fortran-style (order is 'F') contiguous or either one (order is 'A'). Return 0 otherwise. This function always succeeds.

int PyBuffer_ToContiguous(void *buf, Py_buffer *src, Py_ssize_t len, char order)
    Copy len bytes from src to its contiguous representation in buf. order can be 'C' or 'F' (for C-style or Fortran-style ordering). 0 is returned on success, -1 on error.

    This function fails if len != src->len.

void PyBuffer_FillContiguousStrides(int ndims, Py_ssize_t *shape, Py_ssize_t *strides, int itemsize, char order)
    Fill the strides array with byte-strides of a contiguous (C-style if order is 'C' or Fortran-style if order is 'F') array of the given shape with the given number of bytes per element.

int PyBuffer_FillInfo(Py_buffer *view, PyObject *exporter, void *buf, Py_ssize_t len, int readonly, int flags)
    Handle buffer requests for an exporter that wants to expose buf of size len with writability set according to readonly. buf is interpreted as a sequence of unsigned bytes.

    The flags argument indicates the request type. This function always fills in view as specified by flags, unless buf has been designated as read-only and PyBUF_WRITABLE is set in flags.
On success, set view->obj to a new reference to exporter and return 0. Otherwise, raise PyExc_BufferError, set view->obj to NULL and return -1;

If this function is used as part of a getbufferproc, exporter MUST be set to the exporting object and flags must be passed unmodified. Otherwise, exporter MUST be NULL.
Old Buffer Protocol

*Deprecated since version 3.0.*

These functions were part of the “old buffer protocol” API in Python 2. In Python 3, this protocol doesn’t exist anymore but the functions are still exposed to ease porting 2.x code. They act as a compatibility wrapper around the *new buffer protocol*, but they don’t give you control over the lifetime of the resources acquired when a buffer is exported.

Therefore, it is recommended that you call `PyObject_GetBuffer()` (or the `y*` or `w*` format codes with the `PyArg_ParseTuple()` family of functions) to get a buffer view over an object, and `PyBuffer_Release()` when the buffer view can be released.

```c
int PyObject_AsCharBuffer(PyObject *obj, const char **buffer, Py_ssize_t *buffer_len)
```

Returns a pointer to a read-only memory location usable as character-based input. The `obj` argument must support the single-segment character buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error.

```c
int PyObject_AsReadBuffer(PyObject *obj, const void **buffer, Py_ssize_t *buffer_len)
```

Returns a pointer to a read-only memory location containing arbitrary data. The `obj` argument must support the single-segment readable buffer interface. On success, returns 0, sets `buffer` to the memory location and `buffer_len` to the buffer length. Returns -1 and sets a `TypeError` on error.
int **PyObject_CheckReadBuffer**(PyObject *o)

Returns 1 if o supports the single-segment readable buffer interface. Otherwise returns 0. This function always succeeds.

Note that this function tries to get and release a buffer, and exceptions which occur while calling corresponding functions will get suppressed. To get error reporting use **PyObject_GetBuffer()** instead.

int **PyObject_AsWriteBuffer**(PyObject *obj, void **buffer, Py_ssize_t *buffer_len)

Returns a pointer to a writable memory location. The obj argument must support the single-segment, character buffer interface. On success, returns 0, sets buffer to the memory location and buffer_len to the buffer length. Returns -1 and sets a TypeError on error.
Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the “family tree” of Python object types.

**Warning:** While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for *NULL* being passed instead of a valid object. Allowing *NULL* to be passed in can cause memory access violations and immediate termination of the interpreter.
Fundamental Objects

This section describes Python type objects and the singleton object None.

- Type Objects
- The None Object
Numeric Objects

- Integer Objects
- Boolean Objects
- Floating Point Objects
- Complex Number Objects
  - Complex Numbers as C Structures
  - Complex Numbers as Python Objects
Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

- **Bytes Objects**
- **Byte Array Objects**
  - Type check macros
  - Direct API functions
  - Macros
- **Unicode Objects and Codecs**
  - Unicode Objects
    - Unicode Type
    - Unicode Character Properties
    - Creating and accessing Unicode strings
    - Deprecated Py_UNICODE APIs
    - Locale Encoding
    - File System Encoding
    - wchar_t Support
  - Built-in Codecs
    - Generic Codecs
    - UTF-8 Codecs
    - UTF-32 Codecs
    - UTF-16 Codecs
    - UTF-7 Codecs
    - Unicode-Escape Codecs
    - Raw-Unicode-Escape Codecs
    - Latin-1 Codecs
    - ASCII Codecs
    - Character Map Codecs
    - MBCS codecs for Windows
    - Methods & Slots
- Methods and Slot Functions
  - Tuple Objects
  - Struct Sequence Objects
  - List Objects
Container Objects

- Dictionary Objects
- Set Objects
Function Objects

- Function Objects
- Instance Method Objects
- Method Objects
- Cell Objects
- Code Objects
Other Objects

- File Objects
- Module Objects
  - Initializing C modules
    - Single-phase initialization
    - Multi-phase initialization
    - Low-level module creation functions
    - Support functions
  - Module lookup
- Iterator Objects
- Descriptor Objects
- Slice Objects
- Ellipsis Object
- MemoryView objects
- Weak Reference Objects
- Capsules
- Generator Objects
- Coroutine Objects
- Context Variables Objects
- DateTime Objects
Type Objects

PyTypeObject
The C structure of the objects used to describe built-in types.

PyObject* PyType_Type
This is the type object for type objects; it is the same object as type in the Python layer.

int PyType_Check(PyObject *o)
Return true if the object o is a type object, including instances of types derived from the standard type object. Return false in all other cases.

int PyType_CheckExact(PyObject *o)
Return true if the object o is a type object, but not a subtype of the standard type object. Return false in all other cases.

unsigned int PyType_ClearCache()
Clear the internal lookup cache. Return the current version tag.

unsigned long PyType_GetFlags(PyTypeObject* type)
Return the tp_flags member of type. This function is primarily meant for use with Py_LIMITED_API; the individual flag bits are guaranteed to be stable across Python releases, but access to tp_flags itself is not part of the limited API.

New in version 3.2.

Changed in version 3.4: The return type is now unsigned long rather than long.
void **PyType_Modified**(PyTypeObject *type)
Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int **PyType_HasFeature**(PyTypeObject *o, int feature)
Return true if the type object o sets the feature feature. Type features are denoted by single bit flags.

int **PyType_IS_GC**(PyTypeObject *o)
Return true if the type object includes support for the cycle detector; this tests the type flag Py_TPFLAGS_HAVE_GC.

int **PyType_IsSubtype**(PyTypeObject *a, PyTypeObject *b)
Return true if a is a subtype of b.
This function only checks for actual subtypes, which means that __subclasscheck__() is not called on b. Call PyObject_IsSubclass() to do the same check that issubclass() would do.

PyObject* **PyType_GenericAlloc**(PyTypeObject *type,
Py_ssize_t nitems)
*Return value: New reference.*
Generic handler for the tp_alloc slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to NULL.

PyObject* **PyType_GenericNew**(PyTypeObject *type,
PyObject *args, PyObject *kwds)
*Return value: New reference.*
Generic handler for the tp_new slot of a type object. Create a new
instance using the type's `tp_alloc` slot.

```c
int PyType_Ready(PyTypeObject *type)
```
Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type’s base class. Return 0 on success, or return -1 and sets an exception on error.

```c
PyObject* PyType_FromSpec(PyType_Spec *spec)
```
*Return value: New reference.*
Creates and returns a heap type object from the `spec` passed to the function.

```c
PyObject* PyType_FromSpecWithBases(PyType_Spec *spec, PyObject *bases)
```
*Return value: New reference.*
Creates and returns a heap type object from the `spec`. In addition to that, the created heap type contains all types contained by the `bases` tuple as base types. This allows the caller to reference other heap types as base types.

*New in version 3.3.*

```c
void* PyType_GetSlot(PyTypeObject *type, int slot)
```
Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

*New in version 3.4.*
The None Object

Note that the `PyTypeObject` for `None` is not directly exposed in the Python/C API. Since `None` is a singleton, testing for object identity (using `==` in C) is sufficient. There is no `PyNone_Check()` function for the same reason.

`PyObject* Py_None`

The Python `None` object, denoting lack of value. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

`Py_RETURN_NONE`

Properly handle returning `Py_None` from within a C function (that is, increment the reference count of `None` and return it.)
Integer Objects

All integers are implemented as “long” integer objects of arbitrary size.

On error, most PyLong_As* APIs return (return_type)-1 which cannot be distinguished from a number. Use PyErr_Occurred() to disambiguate.

PyLongObject
This subtype of PyObject represents a Python integer object.

PyTypeObject PyLong_Type
This instance of PyTypeObject represents the Python integer type. This is the same object as int in the Python layer.

int PyLong_Check(PyObject *p)
Return true if its argument is a PyLongObject or a subtype of PyLongObject.

int PyLong_CheckExact(PyObject *p)
Return true if its argument is a PyLongObject, but not a subtype of PyLongObject.

PyObject* PyLong_FromLong(long v)
Return a new PyLongObject object from v, or NULL on failure.

The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of 1. I suspect the
behaviour of Python in this case is undefined. :-) 

**PyObject** *PyLong_FromUnsignedLong*(unsigned long *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from a C unsigned long, or *NULL* on failure.

**PyObject** *PyLong_FromSsize_t*(Py_ssize_t *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from a C Py_ssize_t, or *NULL* on failure.

**PyObject** *PyLong_FromSize_t*(size_t *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from a C size_t, or *NULL* on failure.

**PyObject** *PyLong_FromLongLong*(long long *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from a C long long, or *NULL* on failure.

**PyObject** *PyLong_FromUnsignedLongLong*(unsigned long long *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from a C unsigned long long, or *NULL* on failure.

**PyObject** *PyLong_FromDouble*(double *v*)

*Return value: New reference.*
Return a new *PyLongObject* object from the integer part of *v*, or *NULL* on failure.
PyObject* PyLong_FromString(const char *str, char **pend, int base)

*Return value: New reference.*

Return a new PyLongObject based on the string value in `str`, which is interpreted according to the radix in `base`. If `pend` is non-NULL, `*pend` will point to the first character in `str` which follows the representation of the number. If `base` is 0, `str` is interpreted using the Integer literals definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If `base` is not 0, it must be between 2 and 36, inclusive. Leading spaces and single underscores after a base specifier and between digits are ignored. If there are no digits, `ValueError` will be raised.

PyObject* PyLong_FromUnicode(Py_UNICODE *u, Py_ssize_t length, int base)

*Return value: New reference.*

Convert a sequence of Unicode digits to a Python integer value. The Unicode string is first encoded to a byte string using `PyUnicode_EncodeDecimal()` and then converted using `PyLong_FromString()`.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using `PyLong_FromUnicodeObject()`.

PyObject* PyLong_FromUnicodeObject(PyObject *u, int base)

*Return value: New reference.*

Convert a sequence of Unicode digits in the string `u` to a Python integer value. The Unicode string is first encoded to a byte string using `PyUnicode_EncodeDecimal()` and then converted using `PyLong_FromString()`.
New in version 3.3.

**PyObject**\* `PyLong_FromVoidPtr`(void \*p)

*Return value: New reference.*

Create a Python integer from the pointer \( p \). The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

**long** `PyLong_AsLong`(PyObject \*obj)

Return a C long representation of \( obj \). If \( obj \) is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of \( obj \) is out of range for a long.

Returns -1 on error. Use `PyErr_Occurred()` to disambiguate.

**long** `PyLong_AsLongAndOverflow`(PyObject \*obj, int \*overflow)

Return a C long representation of \( obj \). If \( obj \) is not an instance of `PyLongObject`, first call its `__int__()` method (if present) to convert it to a `PyLongObject`.

If the value of \( obj \) is greater than LONG_MAX or less than LONG_MIN, set \*overflow to 1 or -1, respectively, and return -1; otherwise, set \*overflow to 0. If any other exception occurs set \*overflow to 0 and return -1 as usual.

Returns -1 on error. Use `PyErr_Occurred()` to disambiguate.

**long long** `PyLong_AsLongLong`(PyObject \*obj)
Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__() method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a long.

Returns -1 on error. Use `PyErr_Occurred()` to disambiguate.

```c
long long PyLong_AsLongLongAndOverflow(PyObject *obj, int *overflow)
```

Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__int__() method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than PY_LLONG_MAX or less than PY_LLONG_MIN, set *overflow to 1 or -1, respectively, and return -1; otherwise, set *overflow to 0. If any other exception occurs set *overflow to 0 and return -1 as usual.

Returns -1 on error. Use `PyErr_Occurred()` to disambiguate.

New in version 3.2.

```c
Py_ssize_t PyLong_AsSsize_t(PyObject *pylong)
```

Return a C Py_ssize_t representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `Py_ssize_t`.

Returns -1 on error. Use `PyErr_Occurred()` to disambiguate.
unsigned long  **PyLong_AsUnsignedLong** *(PyObject *pylong)*  
Return a C unsigned long representation of *pylong*. *pylong* must be an instance of **PyLongObject**.

Raise **OverflowError** if the value of *pylong* is out of range for a unsigned long.

Returns  

(size_t)-1 on error. Use **PyErr_Occurred()** to disambiguate.

**PyLong_AsSize_t** *(PyObject *pylong)*  
Return a C size_t representation of *pylong*. *pylong* must be an instance of **PyLongObject**.

Raise **OverflowError** if the value of *pylong* is out of range for a size_t.

Returns  

(size_t)-1 on error. Use **PyErr_Occurred()** to disambiguate.

unsigned long long  **PyLong_AsUnsignedLongLong** *(PyObject *pylong)*  
Return a C unsigned long long representation of *pylong*. *pylong* must be an instance of **PyLongObject**.

Raise **OverflowError** if the value of *pylong* is out of range for an unsigned long long.

Returns  

(size_t)-1 on error. Use **PyErr_Occurred()** to disambiguate.

*Changed in version 3.1:* A negative *pylong* now raises **OverflowError**, not **TypeError**.
unsigned long PyLong_AsUnsignedLongMask(PyObject *obj)
Return a C unsigned long representation of obj. If obj is not an instance of PyLongObject, first call its __int__() method (if present) to convert it to a PyLongObject.

If the value of obj is out of range for an unsigned long, return the reduction of that value modulo ULONG_MAX + 1.

Returns -1 on error. Use PyErr_Occurred() to disambiguate.

unsigned long long PyLong_AsUnsignedLongLongMask(PyObject *obj)
Return a C unsigned long long representation of obj. If obj is not an instance of PyLongObject, first call its __int__() method (if present) to convert it to a PyLongObject.

If the value of obj is out of range for an unsigned long long, return the reduction of that value modulo PY_ULLONG_MAX + 1.

Returns -1 on error. Use PyErr_Occurred() to disambiguate.

double PyLong_AsDouble(PyObject *pylong)
Return a C double representation of pylong. pylong must be an instance of PyLongObject.

Raise OverflowError if the value of pylong is out of range for a double.

Returns -1.0 on error. Use PyErr_Occurred() to disambiguate.

void* PyLong_AsVoidPtr(PyObject *pylong)
Convert a Python integer pylong to a C void pointer. If pylong
cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable void pointer for values created with `PyLong_FromVoidPtr()`.

Returns `NULL` on error. Use `PyErr_Occurred()` to disambiguate.
Boolean Objects

Booleans in Python are implemented as a subclass of integers. There are only two booleans, Py_False and Py_True. As such, the normal creation and deletion functions don’t apply to booleans. The following macros are available, however.

`int PyBool_Check(PyObject *o)`

Return true if `o` is of type PyBool_Type.

`PyObject* Py_False`

The Python False object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

`PyObject* Py_True`

The Python True object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

`Py_RETURN_FALSE`

Return Py_False from a function, properly incrementing its reference count.

`Py_RETURN_TRUE`

Return Py_True from a function, properly incrementing its reference count.

`PyObject* PyBool_FromLong(long v)`

_Return value: New reference._

Return a new reference to Py_True or Py_False depending on
the truth value of v.
Floating Point Objects

**PyFloatObject**
This subtype of PyObject represents a Python floating point object.

**PyTypeObject PyFloat_Type**
This instance of PyTypeObject represents the Python floating point type. This is the same object as float in the Python layer.

**int PyFloat_Check(PyObject *p)**
Return true if its argument is a PyFloatObject or a subtype of PyFloatObject.

**int PyFloat_CheckExact(PyObject *p)**
Return true if its argument is a PyFloatObject, but not a subtype of PyFloatObject.

**PyObject* PyFloat_FromString(PyObject *str)**
Create a PyFloatObject object based on the string value in str, or NULL on failure.

**PyObject* PyFloat_FromDouble(double v)**
Create a PyFloatObject object from v, or NULL on failure.

double PyFloat_AsDouble(PyObject *pyfloat)
Return a C double representation of the contents of pyfloat. If pyfloat is not a Python floating point object but has a __float__() method, this method will first be called to convert
`pyfloat` into a float. This method returns `-1.0` upon failure, so one should call `PyErr_Occurred()` to check for errors.

double **PyFloat_AS_DOUBLE**(PyObject *`pyfloat`)
Return a C double representation of the contents of `pyfloat`, but without error checking.

PyObject* **PyFloat_GetInfo**(void)

*Return value: New reference.*
Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It’s a thin wrapper around the header file `float.h`.

double **PyFloat_GetMax**(void)
Return the maximum representable finite float `DBL_MAX` as C double.

double **PyFloat_GetMin**(void)
Return the minimum normalized positive float `DBL_MIN` as C double.

int **PyFloat_ClearFreeList**(void)
Clear the float free list. Return the number of items that could not be freed.
Complex Number Objects

Python’s complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.
Complex Numbers as C Structures

Note that the functions which accept these structures as parameters and return them as results do so by value rather than dereferencing them through pointers. This is consistent throughout the API.

**Py_complex**

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

```c
typedef struct {
    double real;
    double imag;
} Py_complex;
```

**Py_complex Py_c_sum(Py_complex left, Py_complex right)**

Return the sum of two complex numbers, using the C `Py_complex` representation.

**Py_complex Py_c_diff(Py_complex left, Py_complex right)**

Return the difference between two complex numbers, using the C `Py_complex` representation.

**Py_complex Py_c_neg(Py_complex complex)**

Return the negation of the complex number `complex`, using the C `Py_complex` representation.

**Py_complex Py_c_prod(Py_complex left, Py_complex right)**

Return the product of two complex numbers, using the C `Py_complex` representation.
\texttt{Py\_complex \_Py\_c\_quot(Py\_complex dividend, Py\_complex divisor)}

Return the quotient of two complex numbers, using the C \texttt{Py\_complex} representation.

If \textit{divisor} is null, this method returns zero and sets \texttt{errno} to EDOM.

\texttt{Py\_complex \_Py\_c\_pow(Py\_complex num, Py\_complex exp)}

Return the exponentiation of \textit{num} by \textit{exp}, using the C \texttt{Py\_complex} representation.

If \textit{num} is null and \textit{exp} is not a positive real number, this method returns zero and sets \texttt{errno} to EDOM.
Complex Numbers as Python Objects

**PyComplexObject**
This subtype of PyObject represents a Python complex number object.

**PyTypeObject PyComplex_Type**
This instance of PyTypeObject represents the Python complex number type. It is the same object as complex in the Python layer.

int **PyComplex_Check**(PyObject *p)
Return true if its argument is a PyComplexObject or a subtype of PyComplexObject.

int **PyComplex_CheckExact**(PyObject *p)
Return true if its argument is a PyComplexObject, but not a subtype of PyComplexObject.

PyObject* **PyComplex_FromCComplex**(Py_complex v)
*Return value: New reference.*
Create a new Python complex number object from a C Py_complex value.

PyObject* **PyComplex_FromDoubles**(double real, double imag)
*Return value: New reference.*
Return a new PyComplexObject object from real and imag.

double **PyComplex_RealAsDouble**(PyObject *op)
Return the real part of op as a C double.

double **PyComplex_ImagAsDouble**(PyObject *op)
Return the imaginary part of \( op \) as a C double.

\[ \text{Py\_complex} \quad \text{PyComplex\_AsCComplex}(\text{PyObject} *op) \]

Return the \texttt{Py\_complex} value of the complex number \( op \).

If \( op \) is not a Python complex number object but has a \_\_complex\_\_() method, this method will first be called to convert \( op \) to a Python complex number object. Upon failure, this method returns \(-1.0\) as a real value.
Bytes Objects

These functions raise `TypeError` when expecting a bytes parameter and are called with a non-bytes parameter.

**PyBytesObject**

This subtype of `PyObject` represents a Python bytes object.

**PyTypeObject PyBytes_Type**

This instance of `PyTypeObject` represents the Python bytes type; it is the same object as `bytes` in the Python layer.

```c
int PyBytes_Check(PyObject *o)
```

Return true if the object `o` is a bytes object or an instance of a subtype of the bytes type.

```c
int PyBytes_CheckExact(PyObject *o)
```

Return true if the object `o` is a bytes object, but not an instance of a subtype of the bytes type.

```c
PyObject* PyBytes_FromString(const char *v)
```

*Return value: New reference.*

Return a new bytes object with a copy of the string `v` as value on success, and `NULL` on failure. The parameter `v` must not be `NULL`; it will not be checked.

```c
PyObject* PyBytes_FromStringAndSize(const char *v, Py_ssize_t len)
```

*Return value: New reference.*

Return a new bytes object with a copy of the string `v` as value and length `len` on success, and `NULL` on failure. If `v` is `NULL`, the
contents of the bytes object are uninitialized.

**PyObject** `PyBytes_FromFormat`(*const char *format, ...*)

*Return value: New reference.*

Take a C `printf()`-style `format` string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the `format` string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%%</code></td>
<td>n/a</td>
<td>The literal <code>%</code> character.</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>int</td>
<td>A single byte, represented as a C int.</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>int</td>
<td>Equivalent to <code>printf(&quot;%d&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>unsigned int</td>
<td>Equivalent to <code>printf(&quot;%u&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%ld</code></td>
<td>long</td>
<td>Equivalent to <code>printf(&quot;%ld&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%lu</code></td>
<td>unsigned long</td>
<td>Equivalent to <code>printf(&quot;%lu&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%zd</code></td>
<td>Py_ssize_t</td>
<td>Equivalent to <code>printf(&quot;%zd&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%zu</code></td>
<td>size_t</td>
<td>Equivalent to <code>printf(&quot;%zu&quot;)</code>. [1]</td>
</tr>
<tr>
<td><code>%i</code></td>
<td>int</td>
<td>Equivalent to <code>printf(&quot;%i&quot;)</code>. [1]</td>
</tr>
<tr>
<td>Format</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>%%x</td>
<td>int</td>
<td>Equivalent to <code>printf(&quot;%x&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%%s</td>
<td>const char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%%p</td>
<td>const void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to <code>printf(&quot;%p&quot;)</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> yields.</td>
</tr>
</tbody>
</table>

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

(1, 2, 3, 4, 5, 6, 7, 8) For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

**PyObject** *PyBytes_FromFormatV*(const char *format, va_list vargs)

*Return value: New reference.*
Identical to `PyBytes_FromFormat()` except that it takes exactly two arguments.

**PyObject** *PyBytes_FromObject*(PyObject *o)

*Return value: New reference.*
Return the bytes representation of object o that implements the buffer protocol.

**Py_ssize_t** *PyBytes_Size*(PyObject *o)

Return the length of the bytes in bytes object o.
Py_ssize_t **PyBytes_GET_SIZE** *(PyObject *o)*

Macro form of **PyBytes_Size()** but without error checking.

char* **PyBytes_AsString** *(PyObject *o)*

Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of **len(o)** + 1 bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using **PyBytes_FromStringAndSize(NULL, size)**. It must not be deallocated. If *o* is not a bytes object at all, **PyBytes_AsString**() returns **NULL** and raises **TypeError**.

char* **PyBytes_AS_STRING** *(PyObject *string)*

Macro form of **PyBytes_AsString()** but without error checking.

int **PyBytes_AsStringAndSize** *(PyObject *obj, char **buffer, Py_ssize_t *length)*

Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*.

If *length* is **NULL**, the bytes object may not contain embedded null bytes; if it does, the function returns **-1** and a **ValueError** is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using **PyBytes_FromStringAndSize(NULL, size)**. It must not be deallocated. If *obj* is not a bytes object at all, **PyBytes_AsStringAndSize**() returns **-1** and raises **TypeError**.
Changed in version 3.5: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

void **PyBytes_Concat**(PyObject **bytes, PyObject *newpart)

Create a new bytes object in \*bytes containing the contents of \newpart appended to \bytes; the caller will own the new reference. The reference to the old value of \bytes will be stolen. If the new object cannot be created, the old reference to \bytes will still be discarded and the value of \*bytes will be set to NULL; the appropriate exception will be set.

void **PyBytes_ConcatAndDel**(PyObject **bytes, PyObject *newpart)

Create a new bytes object in \*bytes containing the contents of \newpart appended to \bytes. This version decrements the reference count of \newpart.

int _PyBytes_Resize(PyObject **bytes, Py_ssize_t newsize)

A way to resize a bytes object even though it is “immutable”. Only use this to build up a brand new bytes object; don’t use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, \*bytes holds the resized bytes object and 0 is returned; the address in \*bytes may differ from its input value. If the reallocation fails, the original bytes object at \*bytes is deallocated, \*bytes is set to NULL, MemoryError is set, and -1 is returned.
Byte Array Objects

PyByteArrayObject
This subtype of PyObject represents a Python bytearray object.

PyTypeObject PyByteArray_Type
This instance of PyTypeObject represents the Python bytearray type; it is the same object as bytearray in the Python layer.
Type check macros

int **PyByteArray_Check**(PyObject *o)

Return true if the object o is a bytearray object or an instance of a subtype of the bytearray type.

int **PyByteArray_CheckExact**(PyObject *o)

Return true if the object o is a bytearray object, but not an instance of a subtype of the bytearray type.
Direct API functions

**PyObject**\* **PyByteArray\_FromObject**(*PyObject \*o*)  
*Return value: New reference.*  
Return a new bytearray object from any object, o, that implements the buffer protocol.

**PyObject**\* **PyByteArray\_FromStringAndSize**(*const char \*string, Py\_ssize\_t len*)  
*Return value: New reference.*  
Create a new bytearray object from string and its length, len. On failure, NULL is returned.

**PyObject**\* **PyByteArray\_Concat**(*PyObject \*a, PyObject \*b*)  
*Return value: New reference.*  
Concat bytearrays a and b and return a new bytearray with the result.

**Py\_ssize\_t** **PyByteArray\_Size**(*PyObject \*bytearray*)  
Return the size of bytearray after checking for a NULL pointer.

**char**\* **PyByteArray\_AsString**(*PyObject \*bytearray*)  
Return the contents of bytearray as a char array after checking for a NULL pointer. The returned array always has an extra null byte appended.

**int** **PyByteArray\_Resize**(*PyObject \*bytearray, Py\_ssize\_t len*)  
Resize the internal buffer of bytearray to len.
Macros

These macros trade safety for speed and they don’t check pointers.

```c
char* PyByteArray_AS_STRING(PyObject *bytearray)
Macro version of PyByteArray_AsString().
```

```c
Py_ssize_t PyByteArray_GET_SIZE(PyObject *bytearray)
Macro version of PyByteArray_Size().
```
Unicode Objects and Codecs
Unicode Objects

Since the implementation of PEP 393 in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

Py_UNICODE* and UTF-8 representations are created on demand and cached in the Unicode object. The Py_UNICODE* representation is deprecated and inefficient; it should be avoided in performance- or memory-sensitive situations.

Due to the transition between the old APIs and the new APIs, unicode objects can internally be in two states depending on how they were created:

- “canonical” unicode objects are all objects created by a non-deprecated unicode API. They use the most efficient representation allowed by the implementation.
- “legacy” unicode objects have been created through one of the deprecated APIs (typically PyUnicode_FromUnicode()) and only bear the Py_UNICODE* representation; you will have to call PyUnicode_READY() on them before calling any other API.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

Py_UCS4
**Py_UCS2**

**Py_UCS1**
These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use **Py_UCS4**.

*New in version 3.3.*

**Py_UNICODE**
This is a typedef of wchar_t, which is a 16-bit type or 32-bit type depending on the platform.

*Changed in version 3.3:* In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a “narrow” or “wide” Unicode version of Python at build time.

**PyASCIIObject**
**PyCompactUnicodeObject**
**PyUnicodeObject**
These subtypes of **PyObject** represent a Python Unicode object. In almost all cases, they shouldn’t be used directly, since all API functions that deal with Unicode objects take and return **PyObject** pointers.

*New in version 3.3.*

**PyTypeObject** **PyUnicode_Type**
This instance of **PyTypeObject** represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:
int **PyUnicode_Check**(PyObject *)
Return true if the object o is a Unicode object or an instance of a Unicode subtype.

int **PyUnicode_CheckExact**(PyObject *)
Return true if the object o is a Unicode object, but not an instance of a subtype.

int **PyUnicode_READY**(PyObject *)
Ensure the string object o is in the “canonical” representation. This is required before using any of the access macros described below.

Returns 0 on success and -1 with an exception set on failure, which in particular happens if memory allocation fails.

*New in version 3.3.*

Py_ssize_t **PyUnicode_GET_LENGTH**(PyObject *)
Return the length of the Unicode string, in code points. o has to be a Unicode object in the “canonical” representation (not checked).

*New in version 3.3.*

**Py_UCS1** *PyUnicode_1BYTE_DATA*(PyObject *)
**Py_UCS2** *PyUnicode_2BYTE_DATA*(PyObject *)
**Py_UCS4** *PyUnicode_4BYTE_DATA*(PyObject *)
Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use **PyUnicode_KIND()** to select the right macro. Make sure **PyUnicode_READY()** has been called before accessing this.
New in version 3.3.

**PyUnicode_KIND**
- **PyUnicode_WCHAR_KIND**
- **PyUnicode_1BYTE_KIND**
- **PyUnicode_2BYTE_KIND**
- **PyUnicode_4BYTE_KIND**

Return values of the `PyUnicode_KIND()` macro.

New in version 3.3.

```c
int PyUnicode_KIND(PyObject *o)
```

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. `o` has to be a Unicode object in the “canonical” representation (not checked).

New in version 3.3.

```c
void* PyUnicode_DATA(PyObject *o)
```

Return a void pointer to the raw unicode buffer. `o` has to be a Unicode object in the “canonical” representation (not checked).

New in version 3.3.

```c
void PyUnicode_WRITE(int kind, void *data, Pyssize_t index, Py_UCS4 value)
```

Write into a canonical representation `data` (as obtained with `PyUnicode_DATA()`). This macro does not do any sanity checks and is intended for usage in loops. The caller should cache the `kind` value and `data` pointer as obtained from other macro calls. `index` is the index in the string (starts at 0) and `value` is the new code point value which should be written to that location.

New in version 3.3.
**Py_UCS4** `PyUnicode_READ(int kind, void *data, Py_ssize_t index)`

Read a code point from a canonical representation `data` (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

*New in version 3.3.*

**Py_UCS4** `PyUnicode_READ_CHAR(PyObject *o, Py_ssize_t index)`

Read a character from a Unicode object `o`, which must be in the “canonical” representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

*New in version 3.3.*

**PyUnicode_MAX_CHAR_VALUE(PyObject *o)**

Return the maximum code point that is suitable for creating another string based on `o`, which must be in the “canonical” representation. This is always an approximation but more efficient than iterating over the string.

*New in version 3.3.*

**int PyUnicode_ClearFreeList()**

Clear the free list. Return the total number of freed items.

**Py_ssize_t PyUnicode_GET_SIZE(PyObject *o)**

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). `o` has to be a Unicode object (not checked).

*Deprecated since version 3.3, will be removed in version 4.0: Part*
of the old-style Unicode API, please migrate to using
PyUnicode_GET_LENGTH().

Py_ssize_t PyUnicode_GET_DATA_SIZE(PyObject *o)
Return the size of the deprecated Py_UNICODE representation in
bytes. o has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 4.0: Part
of the old-style Unicode API, please migrate to using
PyUnicode_GET_LENGTH().

Py_UNICODE* PyUnicode_AS_UNICODE(PyObject *o)
const char* PyUnicode_AS_DATA(PyObject *o)
Return a pointer to a Py_UNICODE representation of the object.
The returned buffer is always terminated with an extra null code
point. It may also contain embedded null code points, which would
cause the string to be truncated when used in most C functions.
The AS_DATA form casts the pointer to const char *. The o
argument has to be a Unicode object (not checked).

Changed in version 3.3: This macro is now inefficient – because in
many cases the Py_UNICODE representation does not exist and
needs to be created – and can fail (return NULL with an exception
set). Try to port the code to use the new
PyUnicode_nBYTE_DATA() macros or use
PyUnicode_WRITE() or PyUnicode_READ().

Deprecated since version 3.3, will be removed in version 4.0: Part
of the old-style Unicode API, please migrate to using the
PyUnicode_nBYTE_DATA() family of macros.

Unicode Character Properties
Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

```c
int Py_UNICODE_ISSPACE(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a whitespace character.

int Py_UNICODE_ISLOWER(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a lowercase character.

int Py_UNICODE_ISUPPER(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an uppercase character.

int Py_UNICODE_ISTITLE(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a titlecase character.

int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a linebreak character.

int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a digit character.

int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a numeric character.

int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphabetic character.

int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphanumeric
character.

int **Py_UNICODE** **ISPRINTABLE**(Py_UNICODE *ch*)
Return 1 or 0 depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when *repr()* is invoked on a string. It has no bearing on the handling of strings written to *sys.stdout* or *sys.stderr*.)

These APIs can be used for fast direct character conversions:

**Py_UNICODE** **Py_UNICODE** **TOLOWER**(Py_UNICODE *ch*)
Return the character *ch* converted to lower case.

*Deprecated since version 3.3:* This function uses simple case mappings.

**Py_UNICODE** **Py_UNICODE** **TOUPPER**(Py_UNICODE *ch*)
Return the character *ch* converted to upper case.

*Deprecated since version 3.3:* This function uses simple case mappings.

**Py_UNICODE** **Py_UNICODE** **TOTITLE**(Py_UNICODE *ch*)
Return the character *ch* converted to title case.

*Deprecated since version 3.3:* This function uses simple case mappings.

int **Py_UNICODE** **TODECIMAL**(Py_UNICODE *ch*)
Return the character *ch* converted to a decimal positive integer.
Return -1 if this is not possible. This macro does not raise exceptions.

```c
int Py_UNICODE_TODIGIT(Py_UNICODE ch)
    Return the character ch converted to a single digit integer. Return -1 if this is not possible. This macro does not raise exceptions.
```

```c
double Py_UNICODE_TONUMERIC(Py_UNICODE ch)
    Return the character ch converted to a double. Return -1.0 if this is not possible. This macro does not raise exceptions.
```

These APIs can be used to work with surrogates:

```c
Py_UNICODE_IS_SURROGATE(ch)
    Check if ch is a surrogate (0xD800 <= ch <= 0xDBFF).
```

```c
Py_UNICODE_IS_HIGH_SURROGATE(ch)
    Check if ch is a high surrogate (0xD800 <= ch <= 0xDBFF).
```

```c
Py_UNICODE_IS_LOW_SURROGATE(ch)
    Check if ch is a low surrogate (0xDC00 <= ch <= 0xDFFE).
```

```c
Py_UNICODE_JOIN_SURROGATES(high, low)
    Join two surrogate characters and return a single Py_UCS4 value. high and low are respectively the leading and trailing surrogates in a surrogate pair.
```

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

```c
PyObject* PyUnicode_New(Py_ssize_t size, Py_UCS4 maxchar)
```
Create a new Unicode object. maxchar should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

New in version 3.3.

PyObject* PyUnicode_FromKindAndData(int kind, const void *buffer, Py_ssize_t size)

Create a new Unicode object with the given kind (possible values are PyUnicode_1BYTE_KIND etc., as returned by PyUnicode_KIND()). The buffer must point to an array of size units of 1, 2 or 4 bytes per character, as given by the kind.

New in version 3.3.

PyObject* PyUnicode_FromStringAndSize(const char *u, Py_ssize_t size)

Create a Unicode object from the char buffer u. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not NULL, the return value might be a shared object, i.e. modification of the data is not allowed.

If u is NULL, this function behaves like PyUnicode_FromUnicode() with the buffer set to NULL. This usage is deprecated in favor of PyUnicode_New().
PyObject *PyUnicode_FromString(const char *u)

Create a Unicode object from a UTF-8 encoded null-terminated char buffer u.

PyObject* PyUnicode_FromFormat(const char *format, ...)

Take a C printf()-style format string and a variable number of arguments, calculate the size of the resulting Python unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the format ASCII-encoded string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>n/a</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>A single character, represented as a C int.</td>
</tr>
<tr>
<td>%d</td>
<td>int</td>
<td>Equivalent to printf(&quot;%d&quot;).[1]</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned int</td>
<td>Equivalent to printf(&quot;%u&quot;).[1]</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
<td>Equivalent to printf(&quot;%ld&quot;).[1]</td>
</tr>
<tr>
<td>%li</td>
<td>long</td>
<td>Equivalent to printf(&quot;%li&quot;).[1]</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned long</td>
<td>Equivalent to printf(&quot;%lu&quot;).[1]</td>
</tr>
<tr>
<td>Format</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>%lld</td>
<td>long long</td>
<td><code>printf(&quot;%lld&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%lli</td>
<td>long long</td>
<td>Equivalent to <code>printf(&quot;%lli&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%llu</td>
<td>unsigned long long</td>
<td>Equivalent to <code>printf(&quot;%llu&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%zd</td>
<td>Py_ssize_t</td>
<td>Equivalent to <code>printf(&quot;%zd&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%zi</td>
<td>Py_ssize_t</td>
<td>Equivalent to <code>printf(&quot;%zi&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%zu</td>
<td>size_t</td>
<td>Equivalent to <code>printf(&quot;%zu&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>Equivalent to <code>printf(&quot;%i&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%x</td>
<td>int</td>
<td>Equivalent to <code>printf(&quot;%x&quot;)</code>. [1]</td>
</tr>
<tr>
<td>%s</td>
<td>const char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%p</td>
<td>const void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to <code>printf(&quot;%p&quot;)</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> yields.</td>
</tr>
<tr>
<td>%A</td>
<td>PyObject*</td>
<td>The result of calling <code>ascii()</code>.</td>
</tr>
<tr>
<td>%U</td>
<td>PyObject*</td>
<td>A unicode object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A unicode object (which</td>
</tr>
</tbody>
</table>
Python/C API

**%V**  
*PyObject*, *const char*  
May be NULL and a null-terminated C character array as a second parameter (which will be used, if the first parameter is NULL).

**%S**  
*PyObject*  
The result of calling `PyObject_Str()`.

**%R**  
*PyObject*  
The result of calling `PyObject_Repr()`.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

**Note:** The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for "%s" and "%V" (if the *PyObject* argument is NULL), and a number of characters for "%A", "%U", "%S", "%R" and "%V" (if the *PyObject* argument is not NULL).

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13) For integer specifiers [1] (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision is given.

*Changed in version 3.2:* Support for "%lld" and "%llu" added.

*Changed in version 3.3:* Support for "%li", "%lli" and "%zi" added.

*Changed in version 3.4:* Support width and precision formatter for "%s", "%A", "%U", "%V", "%S", "%R" added.

```c
PyObject* PyUnicode_FromFormatV(const char *format,
```
va_list vargs)

Identical to PyUnicode_FromFormat() except that it takes exactly two arguments.

PyObject* PyUnicode_FromEncodedObject(PyObject *obj, const char *encoding, const char *errors)

Decode an encoded object obj to a Unicode object.

bytes, bytearray and other bytes-like objects are decoded according to the given encoding and using the error handling defined by errors. Both can be NULL to have the interface use the default values (see Built-in Codecs for details).

All other objects, including Unicode objects, cause a TypeError to be set.

The API returns NULL if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t PyUnicode_GetLength(PyObject *unicode)

Return the length of the Unicode object, in code points.

New in version 3.3.

Py_ssize_t PyUnicode_CopyCharacters(PyObject *to, Py_ssize_t to_start, PyObject *from, Py_ssize_t from_start, Py_ssize_t how_many)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to memcpy() if possible. Returns -1 and sets an exception on error, otherwise returns the number of copied characters.
New in version 3.3.

Py_ssize_t **PyUnicode_Fill** *(PyObject *unicode, Py_ssize_t start, Py_ssize_t length, Py_UCS4 fill_char)*

Fill a string with a character: write *fill_char* into *unicode[start:start+length]*.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return -1 and raise an exception on error.

New in version 3.3.

int **PyUnicode_WriteChar** *(PyObject *unicode, Py_ssize_t index, Py_UCS4 character)*

Write a character to a string. The string must have been created through **PyUnicode_New()**. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

New in version 3.3.

Py_UCS4 **PyUnicode_ReadChar** *(PyObject *unicode, Py_ssize_t index)*

Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to the macro version **PyUnicode_READ_CHAR()**.
New in version 3.3.

**PyObject** *PyUnicode_Substring**(PyObject *str, Py_ssize_t start, Py_ssize_t end)

*Return value: New reference.*
Return a substring of *str*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

New in version 3.3.

**Py_UCS4** *PyUnicode_AsUCS4**(PyObject *u, Py_UCS4 *buffer, Py_ssize_t buflen, int copy_null)
Copy the string *u* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns NULL and sets an exception on error (in particular, a SystemError if *buflen* is smaller than the length of *u*). *buffer* is returned on success.

New in version 3.3.

**Py_UCS4** *PyUnicode_AsUCS4Copy**(PyObject *u)
Copy the string *u* into a new UCS4 buffer that is allocated using PyMem_Malloc(). If this fails, NULL is returned with a MemoryError set. The returned buffer always has an extra null code point appended.

New in version 3.3.

**Deprecated Py_UNICODE APIs**

*Deprecated since version 3.3, will be removed in version 4.0.*

These API functions are deprecated with the implementation of PEP
Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

**PyObject**

**PyUnicode_FromUnicode**

`PyObject* PyUnicode_FromUnicode(const Py_UNICODE *u, Py_ssize_t size)`

*Return value: New reference.*

Create a Unicode object from the Py_UNICODE buffer `u` of the given size. `u` may be `NULL` which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not `NULL`, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when `u` is `NULL`.

If the buffer is `NULL`, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

Please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()` or `PyUnicode_New()`.

**Py_UNICODE**

**PyUnicode_AsUnicode**

`PyObject* PyUnicode_AsUnicode(PyObject *unicode)`

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, or `NULL` on error. This will create the `Py_UNICODE` representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting `Py_UNICODE` string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Please migrate to using `PyUnicode_AsUCS4()`,
PyUnicode_AsWideChar(), PyUnicode_ReadChar() or similar new APIs.

PyObject*
PyUnicode_TransformDecimalToASCII(Py_UNICODE *s, Py_ssize_t size)

*Return value: New reference.*
Create a Unicode object by replacing all decimal digits in Py_UNICODE buffer of the given size by ASCII digits 0–9 according to their decimal value. Return NULL if an exception occurs.

Py_UNICODE*
PyUnicode_AsUnicodeAndSize(PyObject *unicode, Py_ssize_t *size)

Like PyUnicode_AsUnicode(), but also saves the Py_UNICODE() array length (excluding the extra null terminator) in size. Note that the resulting Py_UNICODE* string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

*New in version 3.3.*

Py_UNICODE*
PyUnicode_AsUnicodeCopy(PyObject *unicode)

Create a copy of a Unicode string ending with a null code point. Return NULL and raise a MemoryError exception on memory allocation failure, otherwise return a new allocated buffer (use PyMem_Free() to free the buffer). Note that the resulting Py_UNICODE* string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

*New in version 3.2.*
Please migrate to using `PyUnicode_AsUCS4Copy()` or similar new APIs.

```c
Py_ssize_t PyUnicode_GetSize(PyObject *unicode)
```

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units).

Please migrate to using `PyUnicode_GetLength()`.

```c
PyObject* PyUnicode_FromObject(PyObject *obj)
```

*Return value: New reference.*

Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If `obj` is already a true Unicode object (not a subtype), return the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a `TypeError`.

### Locale Encoding

The current locale encoding can be used to decode text from the operating system.

```c
PyObject* PyUnicode_DecodeLocaleAndSize(const char *str, Py_ssize_t len, const char *errors)
```

*Return value: New reference.*

Decode a string from UTF-8 on Android, or from the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" *(PEP 383)*. The decoder uses "strict" error handler if `errors` is NULL. `str` must end with a null character but cannot contain embedded null characters.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a
string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

**See also:** The `Py_DecodeLocale()` function.

*New in version 3.3.*

*Changed in version 3.7:* The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_DecodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

```c
PyObject* PyUnicode_DecodeLocale(const char *str, const char *errors)

Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

*New in version 3.3.*

```c
PyObject* PyUnicode_EncodeLocale(PyObject *unicode, const char *errors)

Encode a Unicode object to UTF-8 on Android, or to the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" (PEP 383). The encoder uses "strict" error handler if `errors` is `NULL`. Return a bytes object. `unicode` cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to
Py_FileSystemDefaultEncoding (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

**See also:** The Py_EncodeLocale() function.

*New in version 3.3.*

*Changed in version 3.7:* The function now also uses the current locale encoding for the surrogateescape error handler, except on Android. Previously, Py_EncodeLocale() was used for the surrogateescape, and the current locale encoding was used for strict.

### File System Encoding

To encode and decode file names and other environment strings, Py_FileSystemDefaultEncoding should be used as the encoding, and Py_FileSystemDefaultEncodeErrors should be used as the error handler ([PEP 383](https://www.python.org/dev/peps/pep-0383/) and [PEP 529](https://www.python.org/dev/peps/pep-0529/)). To encode file names to bytes during argument parsing, the "O&" converter should be used, passing PyUnicode_FSConverter() as the conversion function:

```c
int PyUnicode_FSConverter(PyObject* obj, void* result)
```

ParseTuple converter: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using PyUnicode_EncodeFSDefault(); `bytes` objects are output as-is. `result` must be a PyBytesObject* which must be released when it is no longer used.
New in version 3.1.

Changed in version 3.6: Accepts a path-like object.

To decode file names to str during argument parsing, the "0&" converter should be used, passing PyUnicode_FSDecoder() as the conversion function:

```c
int PyUnicode_FSDecoder(PyObject* obj, void* result)
```

ParseTuple converter: decode bytes objects – obtained either directly or indirectly through the os.PathLike interface – to str using PyUnicode_DecodeFSDefaultAndSize(); str objects are output as-is. result must be a PyUnicodeObject* which must be released when it is no longer used.

New in version 3.2.

Changed in version 3.6: Accepts a path-like object.

PyObject* PyUnicode_DecodeFSDefaultAndSize(const char *s, Py_ssize_t size)


Decode a string using Py_FileSystemDefaultEncoding and the Py_FileSystemDefaultEncodeErrors error handler.

If Py_FileSystemDefaultEncoding is not set, fall back to the locale encoding.

Py_FileSystemDefaultEncoding is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use PyUnicode_DecodeLocaleAndSize().
See also: The `Py_DecodeLocale()` function.

In version 3.6:

Use `Py_FileSystemDefaultEncoding` and `Py_FileSystemDefaultEncodeErrors` error handler.

```c
PyObject* PyUnicode_DecodeFSDefault(const char *s)
```

*Return value: New reference.*


If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

In version 3.6:

Use `Py_FileSystemDefaultEncoding` and `Py_FileSystemDefaultEncodeErrors` error handler.

```c
PyObject* PyUnicode_EncodeFSDefault(PyObject *unicode)
```

*Return value: New reference.*

Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return bytes. Note that the resulting bytes object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use
PyUnicode_EncodeLocale().

See also: The Py_EncodeLocale() function.

New in version 3.2.

Changed in version 3.6: Use Py_FileSystemDefaultEncodeErrors error handler.

wchar_t Support

wchar_t support for platforms which support it:

PyObject* PyUnicode_FromWideChar(const wchar_t *w, Py_ssize_t size)

Create a Unicode object from the wchar_t buffer w of the given size. Passing -1 as the size indicates that the function must itself compute the length, using wcslen. Return NULL on failure.

Py_ssize_t PyUnicode_AsWideChar(PyObject *unicode, wchar_t *w, Py_ssize_t size)

Copy the Unicode object contents into the wchar_t buffer w. At most size wchar_t characters are copied (excluding a possibly trailing null termination character). Return the number of wchar_t characters copied or -1 in case of an error. Note that the resulting wchar_t* string may or may not be null-terminated. It is the responsibility of the caller to make sure that the wchar_t* string is null-terminated in case this is required by the application. Also, note that the wchar_t* string might contain null characters, which would cause the string to be truncated when used with most C functions.
wchar_t* **PyUnicode_AsWideCharString**(PyObject *unicode, Py_ssize_t *size)

Convert the Unicode object to a wide character string. The output string always ends with a null character. If `size` is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into `*size`. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If `size` is `NULL` and the `wchar_t`* string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and `*size` is undefined. Raises a `MemoryError` if memory allocation is failed.

*New in version 3.2.*

*Changed in version 3.7:* Raises a `ValueError` if `size` is `NULL` and the `wchar_t`* string contains null characters.
Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is ASCII. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes `setlocale`).

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is “strict” (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

```c
PyObject* PyUnicode_Decode(const char *s, Py_ssize_t size, const char *encoding, const char *errors)
```

_Return value: New reference._
Create a Unicode object by decoding size bytes of the encoded string s. encoding and errors have the same meaning as the parameters of the same name in the str() built-in function. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)

Encode a Unicode object and return the result as Python bytes object. encoding and errors have the same meaning as the parameters of the same name in the Unicode encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_Encode(const Py_UNICODE *s, Py_ssize_t size, const char *encoding, const char *errors)

Encode the Py_UNICODE buffer s of the given size and return a Python bytes object. encoding and errors have the same meaning as the parameters of the same name in the Unicode encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsEncodedString().

UTF-8 Codecs

These are the UTF-8 codec APIs:
PyObject* PyUnicode_DecodeUTF8(const char *s, Py_ssize_t size, const char *errors)

Create a Unicode object by decoding size bytes of the UTF-8 encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecodeUTF8Stateful(const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

If consumed is NULL, behave like PyUnicode_DecodeUTF8(). If consumed is not NULL, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed.

PyObject* PyUnicode_AsUTF8String(PyObject *unicode)

Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

const char* PyUnicode_AsUTF8AndSize(PyObject *unicode, Py_ssize_t *size)

Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in size. The size argument can be NULL; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in size), regardless of whether there are any other null code points.

In the case of an error, NULL is returned with an exception set and no size is stored.
This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer.

*New in version 3.3.*

*Changed in version 3.7:* The return type is now `const char *` rather than `char *`.

```c
const char* PyUnicode_AsUTF8(PyObject *unicode)
```

As `PyUnicode_AsUTF8AndSize()`, but does not store the size.

*New in version 3.3.*

*Changed in version 3.7:* The return type is now `const char *` rather than `char *`.

```c
PyObject* PyUnicode_EncodeUTF8(const Py_UNICODE *s, Py_ssize_t size, const char *errors)
```

*Return value: New reference.*

Encode the `Py_UNICODE` buffer `s` of the given `size` using UTF-8 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF8String()`, `PyUnicode_AsUTF8AndSize()` or `PyUnicode_AsEncodedString()`.

**UTF-32 Codecs**

These are the UTF-32 codec APIs:
PyObject* **PyUnicode_DecodeUTF32** (const char *s,
Py_ssize_t size, const char *errors, int *byteorder)

*Return value: New reference.*

Decode size bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. errors (if non-NULL) defines the error handling. It defaults to “strict”.

If byteorder is non-NULL, the decoder starts decoding using the given byte order:

*byteorder == -1: little endian
*byteorder == 0: native order
*byteorder == 1: big endian

If *byteorder* is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If *byteorder* is -1 or 1, any byte order mark is copied to the output.

After completion, *byteorder* is set to the current byte order at the end of input data.

If byteorder is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF32Stateful** (const char *s,
Py_ssize_t size, const char *errors, int *byteorder,
Py_ssize_t *consumed)

*Return value: New reference.*

If consumed is NULL, behave like **PyUnicode_DecodeUTF32()**. If consumed is not NULL, **PyUnicode_DecodeUTF32Stateful()** will not treat trailing
incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed.

**PyObject**\* **PyUnicode\_AsUTF32String**(**PyObject** \*\*\*\unicode\*)

*Return value: New reference.*

Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

**PyObject**\* **PyUnicode\_EncodeUTF32**(const **Py\_UNICODE** \*\*\s, Py\_ssize\_t \*\*\size, const char \*\*\errors, int \*byteorder)

*Return value: New reference.*

Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

<table>
<thead>
<tr>
<th>byteorder</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>little endian</td>
</tr>
<tr>
<td>0</td>
<td>native byte order (writes a BOM mark)</td>
</tr>
<tr>
<td>1</td>
<td>big endian</td>
</tr>
</tbody>
</table>

If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If **Py\_UNICODE\_WIDE** is not defined, surrogate pairs will be output as a single code point.

Return *NULL* if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part
of the old-style \texttt{Py\_UNICODE} API; please migrate to using \texttt{PyUnicode\_AsUTF32String()} or \texttt{PyUnicode\_AsEncodedString()}.

\textbf{UTF-16 Codecs}

These are the UTF-16 codec APIs:

\begin{verbatim}
PyObject* \textbf{PyUnicode\_DecodeUTF16}(const char *s,
Py_ssize_t size, const char *errors, int *byteorder)

\textit{Return value: New reference.}
\end{verbatim}

Decode \textit{size} bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. \textit{errors} (if non-\textit{NULL}) defines the error handling. It defaults to “strict”.

If \textit{byteorder} is non-\textit{NULL}, the decoder starts decoding using the given byte order:

\begin{itemize}
\item \texttt{\*byteorder == -1}: little endian
\item \texttt{\*byteorder == 0}: native order
\item \texttt{\*byteorder == 1}: big endian
\end{itemize}

If \texttt{\*byteorder} is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If \texttt{\*byteorder} is \texttt{-1} or \texttt{1}, any byte order mark is copied to the output (where it will result in either a \texttt{\ueff} or a \texttt{ufffe} character).

After completion, \texttt{\*byteorder} is set to the current byte order at the end of input data.

If \textit{byteorder} is \textit{NULL}, the codec starts in native order mode.
Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecodeUTF16Stateful(const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

If consumed is NULL, behave like PyUnicode_DecodeUTF16().
If consumed is not NULL, PyUnicode_DecodeUTF16Stateful() will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed.

PyObject* PyUnicode_AsUTF16String(PyObject *unicode)

Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeUTF16(const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in s. Output is written according to the following byte order:

<table>
<thead>
<tr>
<th>byteorder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>little endian</td>
</tr>
<tr>
<td>0</td>
<td>native byte order (writes a BOM)</td>
</tr>
<tr>
<td>1</td>
<td>big endian</td>
</tr>
</tbody>
</table>
If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If Py_UNICODE_WIDE is defined, a single Py_UNICODE value may get represented as a surrogate pair. If it is not defined, each Py_UNICODE values is interpreted as a UCS-2 character.

Return NULL if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsUTF16String() or PyUnicode_AsEncodedString().

**UTF-7 Codecs**

These are the UTF-7 codec APIs:

```c
PyObject* PyUnicode_DecoDEUTF7(const char *s,
Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding size bytes of the UTF-7 encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecoDEUTF7Stateful(const char *s,
Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

*Return value: New reference.*
If consumed is NULL, behave like PyUnicode_DecoDEUTF7(). If consumed is not NULL, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and
the number of bytes that have been decoded will be stored in consumed.

PyObject* **PyUnicode_EncodeUTF7** (const Py_UNICODE *s, Py_ssize_t size, int base64SetO, int base64WhiteSpace, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using UTF-7 and return a Python bytes object. Return NULL if an exception was raised by the codec.

If base64SetO is nonzero, “Set O” (punctuation that has no otherwise special meaning) will be encoded in base-64. If base64WhiteSpace is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python “utf-7” codec.

*Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsEncodedString().*

**Unicode-Escape Codecs**

These are the “Unicode Escape” codec APIs:

PyObject* **PyUnicode_DecodeUnicodeEscape** (const char *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding size bytes of the Unicode-Escape encoded string s. Return NULL if an exception was raised by the codec.

PyObject*
**PyUnicode_AsUnicodeEscapeString** (PyObject *unicode)
Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeUnicodeEscape(const Py_UNICODE* s, Py_ssize_t size)

Encode the Py_UNICODE buffer of the given size using Unicode-Escape and return a bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsUnicodeEscapeString().

Raw-Unicode-Escape Codecs

These are the “Raw Unicode Escape” codec APIs:

PyObject* PyUnicode_DecodeRawUnicodeEscape(const char* s, Py_ssize_t size, const char* errors)

Create a Unicode object by decoding size bytes of the Raw-Unicode-Escape encoded string s. Return NULL if an exception was raised by the codec.

PyObject*
PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)

Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.
PyObject* PyUnicode_EncodeRawUnicodeEscape(const Py_UNICODE *s, Py_ssize_t size)

*Return value: New reference.*

Encode the Py_UNICODE buffer of the given size using Raw-Unicode-Escape and return a bytes object. Return *NULL* if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsRawUnicodeEscapeString() or PyUnicode_AsEncodedString().

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject* PyUnicode_DecodeLatin1(const char *s, Py_ssize_t size, const char *errors)

*Return value: New reference.*

Create a Unicode object by decoding size bytes of the Latin-1 encoded string s. Return *NULL* if an exception was raised by the codec.

PyObject* PyUnicode_AsLatin1String(PyObject *unicode)

*Return value: New reference.*

Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is “strict”. Return *NULL* if an exception was raised by the codec.

PyObject* PyUnicode_EncodeLatin1(const Py_UNICODE *s,
Py_ssize_t size, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using Latin-1 and return a Python bytes object. Return NULL if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsLatin1String() or PyUnicode_AsEncodedString().

**ASCII Codecs**

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject* **PyUnicode_DecodeASCII**(const char *s,
Py_ssize_t size, const char *errors)

*Return value: New reference.*
Create a Unicode object by decoding size bytes of the ASCII encoded string s. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsASCIIString**(PyObject *unicode)

*Return value: New reference.*
Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_EncodeASCII**(const Py_UNICODE *s,
Py_ssize_t size, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using ASCII and return a Python bytes object. Return NULL if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsASCIIString() or PyUnicode_AsEncodedString().

**Character Map Codecs**

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the encodings package). The codec uses mapping to encode and decode characters. The mapping objects provided must support the __getitem__() mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

```c
PyObject* PyUnicode_DecodeCharmap(const char *data, Py_ssize_t size, PyObject *mapping, const char *errors)
```

*Return value: New reference.*

Create a Unicode object by decoding size bytes of the encoded string s using the given mapping object. Return NULL if an exception was raised by the codec.

If mapping is NULL, Latin-1 decoding will be applied. Else mapping must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or None. Unmapped data bytes – ones which cause a LookupError, as well as ones which get mapped to None,
0xFFFE or '\ufffe', are treated as undefined mappings and cause an error.

**PyObject** PyUnicode_AsCharmapString(PyObject *unicode, PyObject *mapping)

*Return value: New reference.*
Encode a Unicode object using the given mapping object and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

The mapping object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or None. Unmapped character ordinals (ones which cause a LookupError) as well as mapped to None are treated as “undefined mapping” and cause an error.

**PyObject** PyUnicode_EncodeCharmap(const Py_UNICODE *s, Py_ssize_t size, PyObject *mapping, const char *errors)

*Return value: New reference.*
Encode the Py_UNICODE buffer of the given size using the given mapping object and return the result as a bytes object. Return NULL if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_AsCharmapString() or PyUnicode_AsEncodedString().

The following codec API is special in that maps Unicode to Unicode.

**PyObject** PyUnicode_Translate(PyObject *unicode, PyObject *mapping, const char *errors)
Translate a Unicode object using the given mapping object and return the resulting Unicode object. Return NULL if an exception was raised by the codec.

The mapping object must map Unicode ordinal integers to Unicode strings, integers (which are then interpreted as Unicode ordinals) or None (causing deletion of the character). Unmapped character ordinals (ones which cause a LookupError) are left untouched and are copied as-is.

PyObject* PyUnicode_TranslateCharmap(const Py_UNICODE*s, Py_ssize_t size, PyObject *mapping, const char *errors)

Translate a Py_UNICODE buffer of the given size by applying a character mapping table to it and return the resulting Unicode object. Return NULL when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Py_UNICODE API; please migrate to using PyUnicode_Translate() or generic codec based API.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject* PyUnicode_DecodeMBCS(const char *s,
Py_ssize_t size, const char *errors)

Create a Unicode object by decoding size bytes of the MBCS encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecodeMBCSStateful(const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

If consumed is NULL, behave like PyUnicode_DecodeMBCS(). If consumed is not NULL, PyUnicode_DecodeMBCSStateful() will not decode trailing lead byte and the number of bytes that have been decoded will be stored in consumed.

PyObject* PyUnicode_AsMBCSString(PyObject *unicode)

Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeCodePage(int code_page, PyObject *unicode, const char *errors)

Encode the Unicode object using the specified code page and return a Python bytes object. Return NULL if an exception was raised by the codec. Use CP_ACP code page to get the MBCS encoder.

New in version 3.3.

PyObject* PyUnicode_EncodeMBCS(const Py_UNICODE *s, Py_ssize_t size, const char *errors)

Encode the `Py_UNICODE` buffer of the given size using MBCS and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

*Deprecated since version 3.3, will be removed in version 4.0:* Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()` or `PyUnicode_AsEncodedString()`.

**Methods & Slots**
Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

**PyObject**

* PyUnicode_Concat(PyObject *left, PyObject *right)
  
  *Return value: New reference.*
  
  Concat two strings giving a new Unicode string.

**PyObject**

* PyUnicode_Split(PyObject *s, PyObject *sep, Py_ssize_t maxsplit)
  
  *Return value: New reference.*
  
  Split a string giving a list of Unicode strings. If `sep` is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most `maxsplit` splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

**PyObject**

* PyUnicode_Splitlines(PyObject *s, int keepend)
  
  *Return value: New reference.*
  
  Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If `keepend` is 0, the Line break characters are not included in the resulting strings.

**PyObject**

* PyUnicode_Translate(PyObject *str, PyObject *table, const char *errors)
  
  Translate a string by applying a character mapping table to it and return the resulting Unicode object.
The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).

Mapping tables need only provide the \texttt{\_\_getitem\_\_()} interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a \texttt{LookupError}) are left untouched and are copied as-is.

\textit{errors} has the usual meaning for codecs. It may be \texttt{NULL} which indicates to use the default error handling.

\texttt{PyObject* \textbf{PyUnicode\_Join}(PyObject *separator, PyObject *seq)}

\textit{Return value: New reference.}

Join a sequence of strings using the given \texttt{separator} and return the resulting Unicode string.

\texttt{Py\_ssize\_t \textbf{PyUnicode\_Tailmatch}(PyObject *str, PyObject *substr, Py\_ssize\_t start, Py\_ssize\_t end, int direction)}

Return 1 if \texttt{substr} matches \texttt{str[start:end]} at the given tail end (\texttt{direction == -1} means to do a prefix match, \texttt{direction == 1} a suffix match), 0 otherwise. Return -1 if an error occurred.

\texttt{Py\_ssize\_t \textbf{PyUnicode\_Find}(PyObject *str, PyObject *substr, Py\_ssize\_t start, Py\_ssize\_t end, int direction)}

Return the first position of \texttt{substr} in \texttt{str[start:end]} using the given \texttt{direction} (\texttt{direction == 1} means to do a forward search, \texttt{direction == -1} a backward search). The return value is the index of the first match; a value of -1 indicates that no match was found, and -2 indicates that an error occurred and an exception has been set.

\texttt{Py\_ssize\_t \textbf{PyUnicode\_FindChar}(PyObject *str, Py\_UCS4 ch,}
Py_ssize_t start, Py_ssize_t end, int direction)
    Return the first position of the character ch in str[start:end]
    using the given direction (direction == 1 means to do a forward
    search, direction == -1 a backward search). The return value is
    the index of the first match; a value of -1 indicates that no match
    was found, and -2 indicates that an error occurred and an
    exception has been set.

    New in version 3.3.

    Changed in version 3.7: start and end are now adjusted to behave
    like str[start:end].

Py_ssize_t PyUnicode_Count(PyObject *str, PyObject *substr,
Py_ssize_t start, Py_ssize_t end)
    Return the number of non-overlapping occurrences of substr in
    str[start:end]. Return -1 if an error occurred.

PyObject* PyUnicode_Replace(PyObject *str, PyObject *substr,
PyObject *replstr, Py_ssize_t maxcount)
    Replace at most maxcount occurrences of substr in str with replstr
    and return the resulting Unicode object. maxcount == -1 means
    replace all occurrences.

int PyUnicode_Compare(PyObject *left, PyObject *right)
    Compare two strings and return -1, 0, 1 for less than, equal, and
    greater than, respectively.

    This function returns -1 upon failure, so one should call
    PyErr_Occurred() to check for errors.

int PyUnicode_CompareWithASCIIString(PyObject *uni,
Compare a unicode object, uni, with string and return -1, 0, 1 for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

```
PyObject* PyUnicode_RichCompare(PyObject *left,
PyObject *right, int op)
```

Rich compare two unicode strings and return one of the following:

- NULL in case an exception was raised
- Py_True or Py_False for successful comparisons
- Py_NotImplemented in case the type combination is unknown

Possible values for op are Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT, and Py_LE.

```
PyObject* PyUnicode_Format(PyObject *format,
PyObject *args)
```

Return a new string object from format and args; this is analogous to `format % args`.

```
int PyUnicode_Contains(PyObject *container,
PyObject *element)
```

Check whether element is contained in container and return true or false accordingly.

element has to coerce to a one element Unicode string. -1 is returned if there was an error.
void **string PyUnicode_InternInPlace(PyObject **string)

Intern the argument *string in place. The argument must be the address of a pointer variable pointing to a Python unicode string object. If there is an existing interned string that is the same as *string, it sets *string to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves *string alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

PyObject* **string PyUnicode_InternFromString(const char *v)

A combination of PyUnicode_FromString() and PyUnicode_InternInPlace(), returning either a new unicode string object that has been interned, or a new (“owned”) reference to an earlier interned string object with the same value.
Tuple Objects

**PyTupleObject**
This subtype of *PyObject* represents a Python tuple object.

**PyTypeObject PyTuple_Type**
This instance of *PyTypeObject* represents the Python tuple type; it is the same object as *tuple* in the Python layer.

int **PyTuple_Check**(PyObject *p)
Return true if *p* is a tuple object or an instance of a subtype of the tuple type.

int **PyTuple_CheckExact**(PyObject *p)
Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type.

*PyObject* **PyTuple_New**(Py_ssize_t len)
*Return value: New reference.*
Return a new tuple object of size *len*, or *NULL* on failure.

*PyObject* **PyTuple_Pack**(Py_ssize_t n, ...)
*Return value: New reference.*
Return a new tuple object of size *n*, or *NULL* on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. *PyTuple_Pack(2, a, b)* is equivalent to *Py_BuildValue("(OO)", a, b)*.

Py_ssize_t **PyTuple_Size**(PyObject *p)
Take a pointer to a tuple object, and return the size of that tuple.
Py_ssize_t **PyTuple_GET_SIZE***(PyObject *p)*
Return the size of the tuple p, which must be non-NULL and point
to a tuple; no error checking is performed.

PyObject* **PyTuple_GetItem***(PyObject *p, Py_ssize_t pos)*
*Return value: Borrowed reference.*
Return the object at position pos in the tuple pointed to by p. If pos
is out of bounds, return NULL and sets an IndexError exception.

PyObject* **PyTuple_GET_ITEM***(PyObject *p, Py_ssize_t pos)*
*Return value: Borrowed reference.*
Like PyTuple_GetItem(), but does no checking of its arguments.

PyObject* **PyTuple_GetSlice***(PyObject *p, Py_ssize_t low,
Py_ssize_t high)*
*Return value: New reference.*
Take a slice of the tuple pointed to by p from low to high and return
it as a new tuple.

int **PyTuple_SetItem***(PyObject *p, Py_ssize_t pos, PyObject *o)*
Insert a reference to object o at position pos of the tuple pointed to
by p. Return 0 on success.

**Note:** This function “steals” a reference to o.

void **PyTuple_SET_ITEM***(PyObject *p, Py_ssize_t pos,
PyObject *o)*
Like PyTuple_SetItem(), but does no error checking, and should only be used to fill in brand new tuples.

**Note:** This function “steals” a reference to o.
int \_PyTuple\_Resize(PyObject \*p, Py_ssize_t newsize)

Can be used to resize a tuple. newsize will be the new length of the tuple. Because tuples are supposed to be immutable, this should only be used if there is only one reference to the object. Do not use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success. Client code should never assume that the resulting value of \*p will be the same as before calling this function. If the object referenced by \*p is replaced, the original \*p is destroyed. On failure, returns -1 and sets \*p to NULL, and raises Memory\_Error or System\_Error.

int PyTuple\_ClearFreeList()

Clear the free list. Return the total number of freed items.
Struct Sequence Objects

Struct sequence objects are the C equivalent of `namedtuple()` objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

```c
void PyStructSequence_InitType(PyTypeObject *type, PyStructSequence_Desc *desc)
    Initializes a struct sequence type `type` from `desc` in place.

int PyStructSequence_InitType2(PyTypeObject *type, PyStructSequence_Desc *desc)
    The same as `PyStructSequence_InitType`, but returns 0 on success and -1 on failure.

New in version 3.4.

PyStructSequence_Desc
    Contains the meta information of a struct sequence type to create.
```

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>const char *</td>
<td>name of the struct sequence</td>
</tr>
<tr>
<td>Field</td>
<td>C Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>doc</td>
<td>const char *</td>
<td>pointer to docstring for the type or NULL to omit</td>
</tr>
<tr>
<td>fields</td>
<td>PyStructSequence_Field *</td>
<td>pointer to NULL-terminated array with field names of the new type</td>
</tr>
<tr>
<td>n_in_sequence</td>
<td>int</td>
<td>number of fields visible to the Python side (if used as tuple)</td>
</tr>
</tbody>
</table>

**PyStructSequence_Field**

Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as `PyObject*`. The index in the fields array of the `PyStructSequence_Desc` determines which field of the struct sequence is described.

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>const char *</td>
<td>name for the field or NULL to end the list of named fields, set to PyStructSequence_UnnamedField to leave unnamed</td>
</tr>
<tr>
<td>doc</td>
<td>const char *</td>
<td>field docstring or NULL to omit</td>
</tr>
</tbody>
</table>
char* **PyStructSequence**\_**UnnamedField**
  Special value for a field name to leave it unnamed.

**PyObject**\* **PyStructSequence**\_**New**(PyTypeObject *\*type)
  *Return value: New reference.*
  Creates an instance of type, which must have been created with
  **PyStructSequence**\_**NewType**().

**PyObject**\* **PyStructSequence**\_**GetItem**(PyObject *\*p,
Py\_ssize\_t pos)
  *Return value: Borrowed reference.*
  Return the object at position pos in the struct sequence pointed to
  by p. No bounds checking is performed.

**PyObject**\* **PyStructSequence**\_**GET\_ITEM**(PyObject *\*p,
Py\_ssize\_t pos)
  *Return value: Borrowed reference.*
  Macro equivalent of **PyStructSequence**\_**GetItem**().

void **PyStructSequence**\_**SetItem**(PyObject *\*p,
Py\_ssize\_t pos, PyObject *\*o)
  Sets the field at index pos of the struct sequence p to value o. Like
  **PyTuple**\_**SET\_ITEM**(), this should only be used to fill in brand
  new instances.

  **Note:** This function “steals” a reference to o.

void **PyStructSequence**\_**SET\_ITEM**(PyObject *\*p,
Py\_ssize\_t *pos, PyObject *\*o)
  Macro equivalent of **PyStructSequence**\_**SetItem**().

  **Note:** This function “steals” a reference to o.
List Objects

**PyListObject**
This subtype of `PyObject` represents a Python list object.

**PyTypeObject PyList_Type**
This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

```c
int PyList_Check(PyObject *p)
```
Return true if `p` is a list object or an instance of a subtype of the list type.

```c
int PyList_CheckExact(PyObject *p)
```
Return true if `p` is a list object, but not an instance of a subtype of the list type.

```c
PyObject* PyList_New(Py_ssize_t len)
```
*Return value: New reference.*
Return a new list of length `len` on success, or `NULL` on failure.

**Note:** If `len` is greater than zero, the returned list object’s items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

```c
Py_ssize_t PyList_Size(PyObject *list)
```
Return the length of the list object in `list`; this is equivalent to `len(list)` on a list object.
Py_ssize_t **PyList_GET_SIZE(PyObject *list)

Macro form of PyList_Size() without error checking.

PyObject* PyList_GetItem(PyObject *list, Py_ssize_t index)

*Return value: Borrowed reference.*
Return the object at position index in the list pointed to by list. The position must be positive, indexing from the end of the list is not supported. If index is out of bounds, return NULL and set an IndexError exception.

PyObject* **PyList_GET_ITEM(PyObject *list, Py_ssize_t i)

*Return value: Borrowed reference.*
Macro form of PyList_GetItem() without error checking.

int int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)

Set the item at index index in list to item. Return 0 on success or -1 on failure.

**Note:** This function “steals” a reference to item and discards a reference to an item already in the list at the affected position.

void PyList_SET_ITEM(PyObject *list, Py_ssize_t i, PyObject *o)

Macro form of PyList_SetItem() without error checking. This is normally only used to fill in new lists where there is no previous content.

**Note:** This macro “steals” a reference to item, and, unlike PyList_SetItem(), does not discard a reference to any item that is being replaced; any reference in list at position i will be leaked.
int **PyList_Insert**(*PyObject *list, Py_ssize_t index, PyObject *item*)

Insert the item *item* into list *list* in front of index *index*. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to *list.insert(index, item)*.

int **PyList_Append**(*PyObject *list, PyObject *item*)

Append the object *item* at the end of list *list*. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to *list.append(item)*.

PyObject**PyList_GetSlice**(*PyObject *list, Py_ssize_t low, Py_ssize_t high*)

*Return value: New reference.*

Return a list of the objects in *list* containing the objects between *low* and *high*. Return NULL and set an exception if unsuccessful. Analogous to *list[low:high]*. Negative indices, as when slicing from Python, are not supported.

int **PyList_SetSlice**(*PyObject *list, Py_ssize_t low, Py_ssize_t high, PyObject *itemlist*)

Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to *list[low:high] = itemlist*. The *itemlist* may be NULL, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure. Negative indices, as when slicing from Python, are not supported.

int **PyList_Sort**(*PyObject *list*)

Sort the items of *list* in place. Return 0 on success, -1 on failure. This is equivalent to *list.sort()*.

int **PyList_Reverse**(*PyObject *list*)
Reverse the items of \textit{list} in place. Return 0 on success, -1 on failure. This is the equivalent of \texttt{list.reverse()}. 

\textbf{PyObject* \texttt{PyList\_AsTuple}(PyObject *\texttt{list})} 

\textit{Return value: New reference.} 
Return a new tuple object containing the contents of \textit{list}; equivalent to \texttt{tuple(list)}. 

\textbf{int \texttt{PyList\_ClearFreeList}()} 

Clear the free list. Return the total number of freed items. 

\textit{New in version 3.3.}
Dictionary Objects

**PyDictObject**

This subtype of **PyObject** represents a Python dictionary object.

**PyTypeObject PyDict_Type**

This instance of **PyTypeObject** represents the Python dictionary type. This is the same object as `dict` in the Python layer.

`int PyDict_Check(PyObject *p)`

Return true if `p` is a dict object or an instance of a subtype of the dict type.

`int PyDict_CheckExact(PyObject *p)`

Return true if `p` is a dict object, but not an instance of a subtype of the dict type.

`PyObject* PyDict_New()`

*Return value: New reference.*

Return a new empty dictionary, or `NULL` on failure.

`PyObject* PyDictProxy_New(PyObject *mapping)`

*Return value: New reference.*

Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

`void PyDict_Clear(PyObject *p)`

Empty an existing dictionary of all key-value pairs.

`int PyDict_Contains(PyObject *p, PyObject *key)`
Determine if dictionary \( p \) contains \( key \). If an item in \( p \) is matches \( key \), return \( 1 \), otherwise return \( 0 \). On error, return \( -1 \). This is equivalent to the Python expression \( \text{key in } p \).

\[ \text{PyObject* PyDict_Copy(PyObject *p)} \]

\[ \text{Return value: New reference.} \]

Return a new dictionary that contains the same key-value pairs as \( p \).

\[ \text{int PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)} \]

Insert \( value \) into the dictionary \( p \) with a key of \( key \). \( key \) must be hashable; if it isn't, \( \text{TypeError} \) will be raised. Return \( 0 \) on success or \( -1 \) on failure.

\[ \text{int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)} \]

Insert \( value \) into the dictionary \( p \) using \( key \) as a key. \( key \) should be a \( \text{const char*} \). The key object is created using \( \text{PyUnicode_FromString(key)} \). Return \( 0 \) on success or \( -1 \) on failure.

\[ \text{int PyDict_DelItem(PyObject *p, PyObject *key)} \]

Remove the entry in dictionary \( p \) with key \( key \). \( key \) must be hashable; if it isn't, \( \text{TypeError} \) is raised. Return \( 0 \) on success or \( -1 \) on failure.

\[ \text{int PyDict_DelItemString(PyObject *p, const char *key)} \]

Remove the entry in dictionary \( p \) which has a key specified by the string \( key \). Return \( 0 \) on success or \( -1 \) on failure.

\[ \text{PyObject* PyDict_GetItem(PyObject *p, PyObject *key)} \]

\[ \text{Return value: Borrowed reference.} \]
Return the object from dictionary $p$ which has a key $key$. Return $NULL$ if the key $key$ is not present, but without setting an exception.

Note that exceptions which occur while calling $\_\_hash\_\_()$ and $\_\_eq\_\_()$ methods will get suppressed. To get error reporting use $\text{PyDict\_GetItemWithError()}$ instead.

**PyObject* $\text{PyDict\_GetItemWithError}$(PyObject *$p$, PyObject *$key$)**

*Return value: Borrowed reference.*

Variant of $\text{PyDict\_GetItem()}$ that does not suppress exceptions. Return $NULL$ with an exception set if an exception occurred. Return $NULL$ without an exception set if the key wasn’t present.

**PyObject* $\text{PyDict\_GetItemString}$(PyObject *$p$, const char *$key$)**

*Return value: Borrowed reference.*

This is the same as $\text{PyDict\_GetItem()}$, but $key$ is specified as a const char*, rather than a PyObject*.

Note that exceptions which occur while calling $\_\_hash\_\_()$ and $\_\_eq\_\_()$ methods and creating a temporary string object will get suppressed. To get error reporting use $\text{PyDict\_GetItemWithError()}$ instead.

**PyObject* $\text{PyDict\_SetDefault}$(PyObject *$p$, PyObject *$key$, PyObject *$defaultobj$)**

*Return value: Borrowed reference.*

This is the same as the Python-level $\text{dict.setdefault()}$. If present, it returns the value corresponding to $key$ from the dictionary $p$. If the key is not in the dict, it is inserted with value
defaultobj and defaultobj is returned. This function evaluates the hash function of key only once, instead of evaluating it independently for the lookup and the insertion.

*New in version 3.4.*

**PyObject**\* **PyDict_Items**(PyObject \*p)

*Return value: New reference.*

Return a PyListObject containing all the items from the dictionary.

**PyObject**\* **PyDict_Keys**(PyObject \*p)

*Return value: New reference.*

Return a PyListObject containing all the keys from the dictionary.

**PyObject**\* **PyDict_Values**(PyObject \*p)

*Return value: New reference.*

Return a PyListObject containing all the values from the dictionary \(p\).

Py\_ssize\_t **PyDict_Size**(PyObject \*p)

Return the number of items in the dictionary. This is equivalent to \(\text{len}(p)\) on a dictionary.

int **PyDict_Next**(PyObject \*p, Py\_ssize\_t \*ppos, PyObject **pkey, PyObject **pvalue)

Iterate over all key-value pairs in the dictionary \(p\). The Py\_ssize\_t referred to by \(ppos\) must be initialized to 0 prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters \(pkey\) and \(pvalue\) should either point to PyObject\* variables that will be filled in with each key
and value, respectively, or may be NULL. Any references returned through them are borrowed. ppos should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

For example:

```python
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value
    /* do something interesting with the values..
    ...

}}

The dictionary p should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```python
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
```
int PyDict_Merge(PyObject *a, PyObject *b, int override)
Iterate over mapping object b adding key-value pairs to dictionary a. b may be a dictionary, or any object supporting PyMapping_Keys() and PyObject_GetItem(). If override is true, existing pairs in a will be replaced if a matching key is found in b, otherwise pairs will only be added if there is not a matching key in a. Return 0 on success or -1 if an exception was raised.

int PyDict_Update(PyObject *a, PyObject *b)
This is the same as PyDict_Merge(a, b, 1) in C, and is similar to a.update(b) in Python except that PyDict_Update() doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no “keys” attribute. Return 0 on success or -1 if an exception was raised.

int PyDict_MergeFromSeq2(PyObject *a, PyObject *seq2, int override)
Update or merge into dictionary a, from the key-value pairs in seq2. seq2 must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if override is true, else the first wins. Return 0 on success or -1 if an exception was raised. Equivalent Python (except for the return value):

```python
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```
int PyDict_ClearFreeList()

Clear the free list. Return the total number of freed items.

*New in version 3.3.*
Set Objects

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using the either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

**PySetObject**

This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

**PyTypeObject** `PySet_Type`

This is an instance of `PyTypeObject` representing the Python `set` type.

**PyTypeObject** `PyFrozenSet_Type`

This is an instance of `PyTypeObject` representing the Python `frozenset` type.
The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

```c
int PySet_Check(PyObject *p)
Return true if p is a set object or an instance of a subtype.
```

```c
int PyFrozenSet_Check(PyObject *p)
Return true if p is a frozenset object or an instance of a subtype.
```

```c
int PyAnySet_Check(PyObject *p)
Return true if p is a set object, a frozenset object, or an instance of a subtype.
```

```c
int PyAnySet_CheckExact(PyObject *p)
Return true if p is a set object or a frozenset object but not an instance of a subtype.
```

```c
int PyFrozenSet_CheckExact(PyObject *p)
Return true if p is a frozenset object but not an instance of a subtype.
```

```c
PyObject* PySet_New(PyObject *iterable)
Return a new set containing objects returned by the iterable. The iterable may be NULL to create a new empty set. Return the new set on success or NULL on failure. Raise TypeError if iterable is not actually iterable. The constructor is also useful for copying a set (c=set(s)).
```

```c
PyObject* PyFrozenSet_New(PyObject *iterable)
```
Return a new `frozenset` containing objects returned by the `iterable`. The `iterable` may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if `iterable` is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

**Py_ssize_t** `PySet_Size(PyObject *anyset)`

Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `PyExc_SystemError` if `anyset` is not a `set`, `frozenset`, or an instance of a subtype.

**Py_ssize_t** `PySet_GET_SIZE(PyObject *anyset)`

Macro form of `PySet_Size()` without error checking.

**int** `PySet_Contains(PyObject *anyset, PyObject *key)`

Return 1 if found, 0 if not found, and -1 if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the `key` is unhashable. Raise `PyExc_SystemError` if `anyset` is not a `set`, `frozenset`, or an instance of a subtype.

**int** `PySet_Add(PyObject *set, PyObject *key)`

Add `key` to a `set` instance. Also works with `frozenset` instances (like `PyTuple_SetItem()` it can be used to fill-in the values of brand new frozensets before they are exposed to other code). Return 0 on success or -1 on failure. Raise a `TypeError` if the `key` is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if `set` is not an instance of `set` or its
subtype.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

```c
int PySet_Discard(PyObject *set, PyObject *key)

// Returns 1 if found and removed, 0 if not found (no action taken),
// and -1 if an error is encountered. Does not raise KeyError for
// missing keys. Raise a TypeError if the key is unhashable. Unlike
// the Python discard() method, this function does not
// automatically convert unhashable sets into temporary frozensets.
// Raise PyExc_SystemError if set is not an instance of set or its
// subtype.
```

```c
PyObject* PySet_Pop(PyObject *set)

// Return a new reference to an arbitrary object in the set, and
// removes the object from the set. Return NULL on failure. Raise
// KeyError if the set is empty. Raise a SystemError if set is not
// an instance of set or its subtype.
```

```c
int PySet_Clear(PyObject *set)

// Empty an existing set of all elements.
```

```c
int PySet_ClearFreeList()

// Clear the free list. Return the total number of freed items.

// New in version 3.3.
```
Function Objects

There are a few functions specific to Python functions.

**PyFunctionObject**

The C structure used for functions.

**PyTypeObject** **PyFunction_Type**

This is an instance of `PyTypeObject` and represents the Python function type. It is exposed to Python programmers as `types.FunctionType`.

```c
int PyFunction_Check(PyObject *o)
```

Return true if `o` is a function object (has type `PyFunction_Type`). The parameter must not be `NULL`.

```c
PyObject* PyFunction_New(PyObject *code, PyObject *globals)
```


Return a new function object associated with the code object `code`. `globals` must be a dictionary with the global variables accessible to the function.

The function’s docstring and name are retrieved from the code object. `__module__` is retrieved from `globals`. The argument defaults, annotations and closure are set to `NULL`. `__qualname__` is set to the same value as the function’s name.

```c
PyObject* PyFunction_NewWithQualName(PyObject *code, PyObject *globals, PyObject *qualname)
```


As `PyFunction_New()`, but also allows setting the function
object’s __qualname__ attribute. _qualname_ should be a unicode object or NULL; if NULL, the __qualname__ attribute is set to the same value as its __name__ attribute.

_New in version 3.3._

**PyObject** *PyFunction_GetCode**(PyObject *op)

_Return value: Borrowed reference._
Return the code object associated with the function object op.

**PyObject** *PyFunction_GetGlobals**(PyObject *op)

_Return value: Borrowed reference._
Return the globals dictionary associated with the function object op.

**PyObject** *PyFunction_GetModule**(PyObject *op)

_Return value: Borrowed reference._
Return the __module__ attribute of the function object op. This is normally a string containing the module name, but can be set to any other object by Python code.

**PyObject** *PyFunction_GetDefaults**(PyObject *op)

_Return value: Borrowed reference._
Return the argument default values of the function object op. This can be a tuple of arguments or NULL.

**int PyFunction_SetDefaults**(PyObject *op, PyObject *defaults)

Set the argument default values for the function object op. defaults must be Py_None or a tuple.

Raises SystemError and returns -1 on failure.
PyObject* PyFunction_GetClosure(PyObject *op)

Return value: Borrowed reference.
Return the closure associated with the function object op. This can be NULL or a tuple of cell objects.

int PyFunction_SetClosure(PyObject *op, PyObject *closure)

Set the closure associated with the function object op. closure must be Py_None or a tuple of cell objects.

Raises SystemError and returns -1 on failure.

PyObject* PyFunction_GetAnnotations(PyObject *op)

Return value: Borrowed reference.
Return the annotations of the function object op. This can be a mutable dictionary or NULL.

int PyFunction_SetAnnotations(PyObject *op, PyObject *annotations)

Set the annotations for the function object op. annotations must be a dictionary or Py_None.

Raises SystemError and returns -1 on failure.
Instance Method Objects

An instance method is a wrapper for a PyCFunction and the new way to bind a PyCFunction to a class object. It replaces the former call PyMethod_New(func, NULL, class).

PyTypeObject PyInstanceMethod_Type
This instance of PyTypeObject represents the Python instance method type. It is not exposed to Python programs.

int PyInstanceMethod_Check(PyObject *o)
Return true if o is an instance method object (has type PyInstanceMethod_Type). The parameter must not be NULL.

PyObject* PyInstanceMethod_New(PyObject *func)
Return a new instance method object, with func being any callable object func is the function that will be called when the instance method is called.

PyObject* PyInstanceMethod_Function(PyObject *im)
Return value: Borrowed reference.
Return the function object associated with the instance method im.

PyObject*
PyInstanceMethod_GET_FUNCTION(PyObject *im)
Return value: Borrowed reference.
Macro version of PyInstanceMethod_Function() which avoids error checking.
Method Objects

Methods are bound function objects. Methods are always bound to an instance of a user-defined class. Unbound methods (methods bound to a class object) are no longer available.

**PyTypeObject PyMethod_Type**

This instance of `PyTypeObject` represents the Python method type. This is exposed to Python programs as `types.MethodType`.

```c
int PyMethod_Check(PyObject *o)
```

Return true if `o` is a method object (has type `PyMethod_Type`). The parameter must not be `NULL`.

```c
PyObject* PyMethod_New(PyObject *func, PyObject *self)
```

*Return value: New reference.*

Return a new method object, with `func` being any callable object and `self` the instance the method should be bound. `func` is the function that will be called when the method is called. `self` must not be `NULL`.

```c
PyObject* PyMethod_Function(PyObject *meth)
```

*Return value: Borrowed reference.*

Return the function object associated with the method `meth`.

```c
PyObject* PyMethod_GET_FUNCTION(PyObject *meth)
```

*Return value: Borrowed reference.*

Macro version of `PyMethod_Function()` which avoids error checking.

```c
PyObject* PyMethod_Self(PyObject *meth)
```
Return value: Borrowed reference.
Return the instance associated with the method meth.

PyObject* PyMethod_GET_SELF(PyObject *meth)
Return value: Borrowed reference.
Macro version of PyMethod_Self() which avoids error checking.

int PyMethod_ClearFreeList()
Clear the free list. Return the total number of freed items.
Cell Objects

“Cell” objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contains a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

**PyCellObject**

The C structure used for cell objects.

**PyTypeObject PyCell_Type**

The type object corresponding to cell objects.

```c
int PyCell_Check(ob)
    Return true if ob is a cell object; ob must not be NULL.
```

```c
PyObject * PyCell_New(PyObject *ob)
    Create and return a new cell object containing the value ob. The parameter may be NULL.
```

```c
PyObject * PyCell_Get(PyObject *cell)
    Return the contents of the cell cell.
```

```c
PyObject * PyCell_GET(PyObject *cell)
    Return value: Borrowed reference.
```
Return the contents of the cell `cell`, but without checking that `cell` is
non-`NULL` and a cell object.

```c
int PyCell_Set(PyObject *cell, PyObject *value)
```
Set the contents of the cell object `cell` to `value`. This releases the
reference to any current content of the cell. `value` may be `NULL`.
`cell` must be non-`NULL`; if it is not a cell object, `-1` will be returned.
On success, `0` will be returned.

```c
void PyCell_SET(PyObject *cell, PyObject *value)
```
Sets the value of the cell object `cell` to `value`. No reference
counts are adjusted, and no checks are made for safety; `cell` must be non-
`NULL` and must be a cell object.
Code Objects

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn’t yet been bound into a function.

**PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

**PyTypeObject PyCode_Type**

This is an instance of PyTypeObject representing the Python code type.

```c
int PyCode_Check(PyObject *co)
```

Return true if `co` is a code object.

```c
int PyCode_GetNumFree(PyCodeObject *co)
```

Return the number of free variables in `co`.

```c
PyCodeObject* PyCode_New(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, PyObject *code, PyObject *consts, PyObject *names, PyObject *varnames, PyObject *freevars, PyObject *cellvars, PyObject *filename, PyObject *name, int firstlineno, PyObject *lnotab)
```

*Return value: New reference.*

Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty()` instead. Calling `PyCode_New()` directly can bind you to a precise Python version since the definition of the bytecode changes often.

```c
PyCodeObject* PyCode_NewEmpty(const char *filename, const
```

Return a new empty code object with the specified filename, function name, and first line number. It is illegal to `exec()` or `eval()` the resulting code object.
File Objects

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (FILE*) support from the C standard library. In Python 3, files and streams use the new io module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter; third-party code is advised to access the io APIs instead.

PyFile_FromFd(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

Create a Python file object from the file descriptor of an already opened file fd. The arguments name, encoding, errors and newline can be NULL to use the defaults; buffering can be -1 to use the default. name is ignored and kept for backward compatibility. Return NULL on failure. For a more comprehensive description of the arguments, please refer to the io.open() function documentation.

Warning: Since Python streams have their own buffering layer, mixing them with OS-level file descriptors can produce various issues (such as unexpected ordering of data).

Changed in version 3.2: Ignore name attribute.

int PyObject_AsFileDescriptor(PyObject *p)

Return the file descriptor associated with p as an int. If the object
is an integer, its value is returned. If not, the object’s `fileno()` method is called if it exists; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

**PyObject** `PyFile_GetLine(PyObject *p, int n)`  
*Return value: New reference.*

Equivalent to `p.readline([n])`, this function reads one line from the object `p`. `p` may be a file object or any object with a `readline()` method. If `n` is 0, exactly one line is read, regardless of the length of the line. If `n` is greater than 0, no more than `n` bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If `n` is less than 0, however, one line is read regardless of length, but `EOFError` is raised if the end of the file is reached immediately.

`int PyFile_WriteObject(PyObject *obj, PyObject *p, int flags)`  
Write object `obj` to file object `p`. The only supported flag for `flags` is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return 0 on success or `-1` on failure; the appropriate exception will be set.

`int PyFile_WriteString(const char *s, PyObject *p)`  
Write string `s` to file object `p`. Return 0 on success or `-1` on failure; the appropriate exception will be set.
Module Objects

**PyTypeObject PyModule_Type**

This instance of *PyTypeObject* represents the Python module type. This is exposed to Python programs as *types.ModuleType*.

```c
int PyModule_Check(PyObject *p)
```

Return true if *p* is a module object, or a subtype of a module object.

```c
int PyModule_CheckExact(PyObject *p)
```

Return true if *p* is a module object, but not a subtype of *PyModule_Type*.

**PyObject* PyModule_NewObject(PyObject *name)**

*Return value: New reference.*

Return a new module object with the **name** attribute set to *name*. The module's **name**, **doc**, **package**, and **loader** attributes are filled in (all but **name** are set to *None*); the caller is responsible for providing a **file** attribute.

*New in version 3.3.*

*Changed in version 3.4:* **package** and **loader** are set to *None*.

**PyObject* PyModule_New(const char *name)**

*Return value: New reference.*

Similar to *PyModule_NewObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.
PyObject* **PyModule_GetDict**(PyObject *module)

*Return value: Borrowed reference.*
Return the dictionary object that implements *module*'s namespace; this object is the same as the **__dict__** attribute of the module object. If *module* is not a module object (or a subtype of a module object), **SystemError** is raised and **NULL** is returned.

It is recommended extensions use other **PyModule_*()** and **PyObject_*()** functions rather than directly manipulate a module's **__dict__**.

PyObject* **PyModule_GetNameObject**(PyObject *module)

*Return value: New reference.*
Return *module*'s **__name__** value. If the module does not provide one, or if it is not a string, **SystemError** is raised and **NULL** is returned.

*New in version 3.3.*

const char* **PyModule_GetName**(PyObject *module)

Similar to **PyModule_GetNameObject()** but return the name encoded to 'utf-8'.

void* **PyModule_GetState**(PyObject *module)

Return the “state” of the module, that is, a pointer to the block of memory allocated at module creation time, or **NULL**. See **PyModuleDef.m_size**.

**PyModuleDef*** **PyModule_GetDef**(PyObject *module)

Return a pointer to the **PyModuleDef** struct from which the module was created, or **NULL** if the module wasn’t created from a definition.
PyObject*
PyModule_GetFilenameObject(PyObject *module)

Return the name of the file from which module was loaded using module’s __file__ attribute. If this is not defined, or if it is not a unicode string, raise SystemError and return NULL; otherwise return a reference to a Unicode object.

New in version 3.2.

const char* PyModule_GetFilename(PyObject *module)

Similar to PyModule_GetFilenameObject() but return the filename encoded to ‘utf-8’.

Deprecated since version 3.2: PyModule_GetFilename() raises UnicodeEncodeError on unencodable filenames, use PyModule_GetFilenameObject() instead.
Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using `PyImport_AppendInittab()`). See Building C and C++ Extensions or Extending Embedded Python for details.

The initialization function can either pass a module definition instance to `PyModule_Create()`, and return the resulting module object, or request “multi-phase initialization” by returning the definition struct itself.

**PyModuleDef**

The module definition struct, which holds all information needed to create a module object. There is usually only one statically initialized variable of this type for each module.

PyModuleDef_Base m_base

Always initialize this member to `PyModuleDef_HEAD_INIT`.

const char *m_name

Name for the new module.

const char *m_doc

Docstring for the module; usually a docstring variable created with `PyDoc_STRVAR()` is used.

Py_ssize_t m_size

Module state may be kept in a per-module memory area that can be retrieved with `PyModule_GetState()`, rather than in static globals. This makes modules safe for use in multiple sub-
interpreters.

This memory area is allocated based on \texttt{m\_size} on module creation, and freed when the module object is deallocated, after the \texttt{m\_free} function has been called, if present.

Setting \texttt{m\_size} to \texttt{-1} means that the module does not support sub-interpreters, because it has global state.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state. Non-negative \texttt{m\_size} is required for multi-phase initialization.

See \texttt{PEP 3121} for more details.

\texttt{PyMethodDef* m\_methods}

A pointer to a table of module-level functions, described by \texttt{PyMethodDef} values. Can be \texttt{NULL} if no functions are present.

\texttt{PyModuleDef\_Slot* m\_slots}

An array of slot definitions for multi-phase initialization, terminated by a \texttt{\{0, NULL\}} entry. When using single-phase initialization, \texttt{m\_slots} must be \texttt{NULL}.

\textit{Changed in version 3.5}: Prior to version 3.5, this member was always set to \texttt{NULL}, and was defined as:

\begin{verbatim}
  inquiry m\_reload
\end{verbatim}

\texttt{traverseproc m\_traverse}

A traversal function to call during GC traversal of the module object, or \texttt{NULL} if not needed. This function may be called
before module state is allocated (\texttt{PyModule\_GetState()} may return \texttt{NULL}), and before the \texttt{Py\_mod\_exec} function is executed.

\textbf{inquiry \texttt{m\_clear}}

A clear function to call during GC clearing of the module object, or \texttt{NULL} if not needed. This function may be called before module state is allocated (\texttt{PyModule\_GetState()} may return \texttt{NULL}), and before the \texttt{Py\_mod\_exec} function is executed.

\textbf{freefunc \texttt{m\_free}}

A function to call during deallocation of the module object, or \texttt{NULL} if not needed. This function may be called before module state is allocated (\texttt{PyModule\_GetState()} may return \texttt{NULL}), and before the \texttt{Py\_mod\_exec} function is executed.

\section*{Single-phase initialization}

The module initialization function may create and return the module object directly. This is referred to as “single-phase initialization”, and uses one of the following two module creation functions:

\begin{verbatim}
PyObject* PyModule_Create(PyModuleDef *def)
    \textit{Return value: New reference.}
    Create a new module object, given the definition in \texttt{def}. This behaves like \texttt{PyModule\_Create2()} with \texttt{module\_api\_version} set to PYTHON\_API\_VERSION.

PyObject* PyModule_Create2(PyModuleDef *def, int module_api_version)
    \textit{Return value: New reference.}
    Create a new module object, given the definition in \texttt{def}, assuming
\end{verbatim}
the API version `module_api_version`. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

**Note:** Most uses of this function should be using `PyModule_Create()` instead; only use this if you are sure you need it.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like `PyModule_AddObject()`.

**Multi-phase initialization**

An alternate way to specify extensions is to request “multi-phase initialization”. Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the `sys.modules` entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection – as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using `PyModule_GetState()`), or its contents (such as the module's `__dict__` or individual classes created with `PyType_FromSpec()`).
All modules created using multi-phase initialization are expected to support sub-interpreters. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a PyModuleDef instance with non-empty m_slots. Before it is returned, the PyModuleDef instance must be initialized with the following function:

```c
PyObject* PyModuleDef_Init(PyModuleDef *def)
```

*Return value: Borrowed reference.*

Ensures a module definition is a properly initialized Python object that correctly reports its type and reference count.

Returns def cast to PyObject*, or NULL if an error occurred.

*New in version 3.5.*

The m_slots member of the module definition must point to an array of PyModuleDef_Slot structures:

```c
PyModuleDef_Slot
```

```c
int slot
```

A slot ID, chosen from the available values explained below.

```c
void* value
```

Value of the slot, whose meaning depends on the slot ID.

*New in version 3.5.*

The m_slots array must be terminated by a slot with id 0.

The available slot types are:
**Py_mod_create**

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

```c
PyObject* create_module(PyObject *spec, PyModuleDef *def)
```

The function receives a *ModuleSpec* instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return *NULL*.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using `PyModule_New()`. The name is taken from `spec`, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of `PyModule_Type`. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

---

**Note:**

- The Python documentation uses the term `PyObject` instead of the C type `PyObject*`.
- The examples in the documentation use the `PyModuleDef` struct, which is not part of the Python API but is used to define a module.
- The `PyModuleDef` struct is not mentioned in the documentation, but it is used to define a module.
- The `PyModule_Type` is the type of a module object in Python.
- The `PyModule_New()` function is used to create a normal module object.
- The `PyModuleDef` struct is used to define a module.
- The `PyModule_Type` instances may only be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.
**Py_mod_exec**

Specifies a function that is called to execute the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

```
int exec_module(PyObject* module)
```

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the `m_slots` array.

See PEP 489 for more details on multi-phase initialization.

### Low-level module creation functions

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

**PyObject * PyModule_FromDefAndSpec** *(PyModuleDef *def, PyObject *spec)*

*Return value: New reference.*

Create a new module object, given the definition in `module` and the ModuleSpec `spec`. This behaves like `PyModule_FromDefAndSpec2()` with `module_api_version` set to `PYTHON_API_VERSION`.

*New in version 3.5.*

**PyObject * PyModule_FromDefAndSpec2** *(PyModuleDef *def, PyObject *spec, int module_api_version)*
Create a new module object, given the definition in module and the ModuleSpec spec, assuming the API version module_api_version. If that version does not match the version of the running interpreter, a RuntimeWarning is emitted.

**Note:** Most uses of this function should be using PyModule_FromDefAndSpec() instead; only use this if you are sure you need it.

*New in version 3.5.*

```c
int PyModule_ExecDef(PyObject *module, PyModuleDef *def)
```
Process any execution slots (Py_mod_exec) given in def.

*New in version 3.5.*

```c
int PyModule_SetDocString(PyObject *module, const char *docstring)
```
Set the docstring for module to docstring. This function is called automatically when creating a module from PyModuleDef, using either PyModule_Create or PyModule_FromDefAndSpec.

*New in version 3.5.*

```c
int PyModule_AddFunctions(PyObject *module, PyMethodDef *functions)
```
Add the functions from the NULL terminated functions array to module. Refer to the PyMethodDef documentation for details on individual entries (due to the lack of a shared module namespace, module level “functions” implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically.
when creating a module from PyModuleDef, using either PyModule_Create or PyModule_FromDefAndSpec.

*New in version 3.5.*

**Support functions**

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state:

```c
int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)
```

Add an object to `module` as `name`. This is a convenience function which can be used from the module's initialization function. This steals a reference to `value`. Return -1 on error, 0 on success.

```c
int PyModule_AddIntConstant(PyObject *module, const char *name, long value)
```

Add an integer constant to `module` as `name`. This convenience function can be used from the module's initialization function. Return -1 on error, 0 on success.

```c
int PyModule_AddStringConstant(PyObject *module, const char *name, const char *value)
```

Add a string constant to `module` as `name`. This convenience function can be used from the module's initialization function. The string `value` must be NULL-terminated. Return -1 on error, 0 on success.

```c
int PyModule_AddIntMacro(PyObject *module, macro)
```
Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant `AF_INET` with the value of `AF_INET` to *module*. Return `-1` on error, `0` on success.

```c
int PyModule_AddStringMacro(PyObject *module, macro)
Add a string constant to *module*.
```
Module lookup

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

**PyObject**\* **PyState\_FindModule**\( (\text{PyModuleDef} \* \text{def}) \)  
*Return value: Borrowed reference.*  
Returns the module object that was created from `def` for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState\_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

**int** **PyState\_AddModule**\( (\text{PyObject} \* \text{module}, \text{PyModuleDef} \* \text{def}) \)  
Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState\_FindModule()`.

Only effective on modules created using single-phase initialization.

*New in version 3.3.*

**int** **PyState\_RemoveModule**\( (\text{PyModuleDef} \* \text{def}) \)  
Removes the module object created from `def` from the interpreter state.
Iterator Objects

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the `__getitem__()` method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

**PyTypeObject PySeqIter_Type**
Type object for iterator objects returned by `PySeqIter_New()` and the one-argument form of the `iter()` built-in function for built-in sequence types.

**int PySeqIter_Check(op)**
Return true if the type of `op` is `PySeqIter_Type`.

**PyObject* PySeqIter_New(PyObject *seq)**
*Return value: New reference.*
Return an iterator that works with a general sequence object, `seq`. The iteration ends when the sequence raises `IndexError` for the subscripting operation.

**PyTypeObject PyCallIter_Type**
Type object for iterator objects returned by `PyCallIter_New()` and the two-argument form of the `iter()` built-in function.

**int PyCallIter_Check(op)**
Return true if the type of `op` is `PyCallIter_Type`.

**PyObject* PyCallIter_New(PyObject *callable, PyObject *sentinel)**
**Return value: New reference.**

Return a new iterator. The first parameter, `callable`, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When `callable` returns a value equal to `sentinel`, the iteration will be terminated.
Descriptor Objects

“Descriptors” are objects that describe some attribute of an object. They are found in the dictionary of type objects.

`PyTypeObject PyProperty_Type`

The type object for the built-in descriptor types.

`PyObject* PyDescr_NewGetSet(PyTypeObject *type, struct PyGetSetDef *getset)`

*Return value: New reference.*

`PyObject* PyDescr_NewMember(PyTypeObject *type, struct PyMemberDef *meth)`

*Return value: New reference.*

`PyObject* PyDescr_NewMethod(PyTypeObject *type, struct PyMethodDef *meth)`

*Return value: New reference.*

`PyObject* PyDescr_NewWrapper(PyTypeObject *type, struct wrapperbase *wrapper, void *wrapped)`

*Return value: New reference.*

`PyObject* PyDescr_NewClassMethod(PyTypeObject *type, PyMethodDef *method)`

*Return value: New reference.*

`int PyDescr_IsData(PyObject *descr)`

Return true if the descriptor objects `descr` describes a data attribute, or false if it describes a method. `descr` must be a descriptor object; there is no error checking.
PyObject* PyWrapper_New(PyObject *, PyObject *)

Slice Objects

PyTypeObject **PySlice_Type**
The type object for slice objects. This is the same as slice in the Python layer.

int **PySlice_Check**(PyObject *ob)
Return true if ob is a slice object; ob must not be NULL.

PyObject* **PySlice_New**(PyObject *start, PyObject *stop, PyObject *step)
*Return value: New reference.*
Return a new slice object with the given values. The start, stop, and step parameters are used as the values of the slice object attributes of the same names. Any of the values may be NULL, in which case the None will be used for the corresponding attribute. Return NULL if the new object could not be allocated.

int **PySlice_GetIndices**(PyObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)
Retrieve the start, stop and step indices from the slice object slice, assuming a sequence of length length. Treats indices greater than length as errors.

Returns 0 on success and -1 on error with no exception set (unless one of the indices was not None and failed to be converted to an integer, in which case -1 is returned with an exception set).

You probably do not want to use this function.

*Changed in version 3.2:* The parameter type for the slice parameter
int **PySlice_GetIndicesEx**(PyObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step, Py_ssize_t *slicelength)

Usable replacement for **PySlice_GetIndices()**. Retrieve the start, stop, and step indices from the slice object \textit{slice} assuming a sequence of length \textit{length}, and store the length of the slice in \textit{slicelength}. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns 0 on success and -1 on error with exception set.

\textbf{Note:} This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of **PySlice_Unpack()** and **PySlice_AdjustIndices()** where

```python
if (PySlice_GetIndicesEx(slice, length, &start, // return error
    } }
```

is replaced by

```python
if (PySlice_Unpack(slice, &start, &stop, &step) // return error
} }
```

\textit{Changed in version 3.2:} The parameter type for the \textit{slice} parameter was \texttt{PySliceObject*} before.

\textit{Changed in version 3.6.1:} If Py\_LIMITED\_API is not set or set to the value between 0x03050400 and 0x03060000 (not including)
or 0x03060100 or higher PySlice_GetIndicesEx() is implemented as a macro using PySlice_Unpack() and PySlice_AdjustIndices(). Arguments start, stop and step are evaluated more than once.

*Deprecated since version 3.6.1:* If Py_LIMITED_API is set to the value less than 0x03050400 or between 0x03060000 and 0x03060100 (not including) PySlice_GetIndicesEx() is a deprecated function.

```c
int PySlice_Unpack(PyObject *slice, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)
```

Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than PY_SSIZE_T_MAX to PY_SSIZE_T_MAX, silently boost the start and stop values less than PY_SSIZE_T_MIN to PY_SSIZE_T_MIN, and silently boost the step values less than -PY_SSIZE_T_MAX to -PY_SSIZE_T_MAX.

Return -1 on error, 0 on success.

*New in version 3.6.1.*

```c
Py_ssize_t PySlice_AdjustIndices(Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t step)
```

Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn’t call Python code.

*New in version 3.6.1.*
Ellipsis Object

PyObject *Py_Ellipsis

The Python Ellipsis object. This object has no methods. It needs to be treated just like any other object with respect to reference counts. Like Py_None it is a singleton object.
MemoryView objects

A `memoryview` object exposes the C level `buffer interface` as a Python object which can then be passed around like any other object.

`PyObject *PyMemoryView_FromObject(PyObject *obj)`

Create a memoryview object from an object that provides the buffer interface. If `obj` supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

`PyObject *PyMemoryView_FromMemory(char *mem, Py_ssize_t size, int flags)`

Create a memoryview object using `mem` as the underlying buffer. `flags` can be one of `PyBUF_READ` or `PyBUF_WRITE`.

*New in version 3.3.*

`PyObject *PyMemoryView_FromBuffer(Py_buffer *view)`

Create a memoryview object wrapping the given buffer structure `view`. For simple byte buffers, `PyMemoryView_FromMemory()` is the preferred function.

`PyObject *PyMemoryView_GetContiguous(PyObject *obj, int buffertype, char order)`

Create a memoryview object to a `contiguous` chunk of memory (in either ‘C’ or ‘F’ortran order) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to
the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

```c
int PyMemoryView_Check(PyObject *obj)
```

Return true if the object `obj` is a memoryview object. It is not currently allowed to create subclasses of `memoryview`.

```c
Py_buffer *PyMemoryView_GET_BUFFER(PyObject *mview)
```

Return a pointer to the memoryview's private copy of the exporter's buffer. `mview` must be a memoryview instance; this macro doesn’t check its type, you must do it yourself or you will risk crashes.

```c
Py_buffer *PyMemoryView_GET_BASE(PyObject *mview)
```

Return either a pointer to the exporting object that the memoryview is based on or `NULL` if the memoryview has been created by one of the functions `PyMemoryView_FromMemory()` or `PyMemoryView_FromBuffer()`. `mview` must be a memoryview instance.
Weak Reference Objects

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

```c
int PyWeakref_Check(ob)
    Return true if ob is either a reference or proxy object.
```

```c
int PyWeakref_CheckRef(ob)
    Return true if ob is a reference object.
```

```c
int PyWeakref_CheckProxy(ob)
    Return true if ob is a proxy object.
```

```c
PyObject* PyWeakref_NewRef(PyObject *ob,
PyObject *callback)
    Return a weak reference object for the object ob. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, callback, can be a callable object that receives notification when ob is garbage collected; it should accept a single parameter, which will be the weak reference object itself. callback may also be None or NULL. If ob is not a weakly-referencable object, or if callback is not callable, None, or NULL, this will return NULL and raise TypeError.
```

```c
PyObject* PyWeakref_NewProxy(PyObject *ob,
PyObject *callback)
```
Return a weak reference proxy object for the object `ob`. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, `callback`, can be a callable object that receives notification when `ob` is garbage collected; it should accept a single parameter, which will be the weak reference object itself. `callback` may also be `None` or `NULL`. If `ob` is not a weakly-referencable object, or if `callback` is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

```python
PyObject* PyWeakref_GetObject(PyObject *ref)
```

Return value: Borrowed reference.
Return the referenced object from a weak reference, `ref`. If the referent is no longer live, returns `Py_None`.

**Note:** This function returns a **borrowed reference** to the referenced object. This means that you should always call `Py_INCREF()` on the object except if you know that it cannot be destroyed while you are still using it.

```python
PyObject* PyWeakref_GET_OBJECT(PyObject *ref)
```

Return value: Borrowed reference.
Similar to `PyWeakref_GetObject()`, but implemented as a macro that does no error checking.
Capsules

Refer to Providing a C API for an Extension Module for more information on using these objects.

*New in version 3.1.*

**PyCapsule**

This subtype of PyObject represents an opaque value, useful for C extension modules who need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

**PyCapsuleDestructor**

The type of a destructor callback for a capsule. Defined as:

```c
typedef void (*PyCapsuleDestructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsuleDestructor callbacks.

```c
int PyCapsule_CheckExact(PyObject *p)
```

Return true if its argument is a PyCapsule.

```c
PyObject* PyCapsule_New(void *pointer, const char *name, PyCapsuleDestructor destructor)

Create a PyCapsule encapsulating the `pointer`. The `pointer` argument may not be `NULL`.```
On failure, set an exception and return *NULL*.

The *name* string may either be *NULL* or a pointer to a valid C string. If non-*NULL*, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not *NULL*, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as modulename.attributename. This will enable other modules to import the capsule using PyCapsule_Import().

```c
void* PyCapsule_GetPointer(PyObject *capsule, const char *name)
```

Retrieve the *pointer* stored in the capsule. On failure, set an exception and return *NULL*.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is *NULL*, the *name* passed in must also be *NULL*. Python uses the C function strcmp() to compare capsule names.

```c
PyCapsule_Destructor
PyCapsule_GetDestructor(PyObject *capsule)
```

Return the current destructor stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* destructor. This makes a *NULL* return code somewhat ambiguous; use PyCapsule_IsValid() or PyErr_Occurred() to disambiguate.
void* **PyCapsule_GetContext**(PyObject *capsule)

Return the current context stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* context. This makes a *NULL* return code somewhat ambiguous; use **PyCapsule_IsValid()** or **PyErr_Occurred()** to disambiguate.

const char* **PyCapsule_GetName**(PyObject *capsule)

Return the current name stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* name. This makes a *NULL* return code somewhat ambiguous; use **PyCapsule_IsValid()** or **PyErr_Occurred()** to disambiguate.

void* **PyCapsule_Import**(const char *name, int no_block)

Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly. If *no_block* is true, import the module without blocking (using **PyImport_ImportModuleNoBlock()**). If *no_block* is false, import the module conventionally (using **PyImport_ImportModule()**).

Return the capsule’s internal *pointer* on success. On failure, set an exception and return *NULL*.

int **PyCapsule_IsValid**(PyObject *capsule, const char *name)

Determines whether or not *capsule* is a valid capsule. A valid capsule is non-*NULL*, passes **PyCapsule_CheckExact()**, has a
non-NULL pointer stored in it, and its internal name matches the name parameter. (See PyCapsule_GetPointer() for information on how capsule names are compared.)

In other words, if PyCapsule_IsValid() returns a true value, calls to any of the accessors (any function starting with PyCapsule_Get()) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return 0 otherwise. This function will not fail.

int PyCapsule_SetContext(PyObject *capsule, void *context)
Set the context pointer inside capsule to context.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetDestructor(PyObject *capsule, PyCapsule_Destructor destructor)
Set the destructor inside capsule to destructor.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetName(PyObject *capsule, const char *name)
Set the name inside capsule to name. If non-NULL, the name must outlive the capsule. If the previous name stored in the capsule was not NULL, no attempt is made to free it.

Return 0 on success. Return nonzero and set an exception on failure.

int PyCapsule_SetPointer(PyObject *capsule, void *pointer)
Set the void pointer inside *capsule to pointer*. The pointer may not be *NULL*.

Return 0 on success. Return nonzero and set an exception on failure.
Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling `PyGen_New()` or `PyGen_NewWithQualName()`.

**PyGenObject**

The C structure used for generator objects.

**PyTypeObject PyGen_Type**

The type object corresponding to generator objects.

```c
int PyGen_Check(PyObject *ob)
Return true if ob is a generator object; ob must not be NULL.
```

```c
int PyGen_CheckExact(PyObject *ob)
Return true if ob's type is PyGen_Type; ob must not be NULL.
```

```c
PyObject* PyGen_New(PyFrameObject *frame)
Create and return a new generator object based on the frame object. A reference to frame is stolen by this function. The argument must not be NULL.
```

```c
PyObject* PyGen_NewWithQualName(PyFrameObject *frame, PyObject *name, PyObject *qualname)
Create and return a new generator object based on the frame object, with __name__ and __qualname__ set to name and qualname. A reference to frame is stolen by this function. The
frame argument must not be NULL.
Coroutine Objects

*New in version 3.5.*

Coroutine objects are what functions declared with an `async` keyword return.

**PyCoroObject**

The C structure used for coroutine objects.

**PyTypeObject PyCoro_Type**

The type object corresponding to coroutine objects.

```c
int PyCoro_CheckExact(PyObject *ob)
```

Return true if `ob`'s type is `PyCoro_Type`; `ob` must not be `NULL`.

```c
PyObject* PyCoro_New(PyFrameObject *frame, PyObject *name, PyObject *qualname)
```

*Return value: New reference.*

Create and return a new coroutine object based on the `frame` object, with `__name__` and `__qualname__` set to `name` and `qualname`. A reference to `frame` is stolen by this function. The `frame` argument must not be `NULL`.

© Copyright 2001-2019, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. Please donate.

Last updated on Mar 25, 2019. Found a bug?

Created using Sphinx 1.8.5.
Context Variables Objects

Note: Changed in version 3.7.1:

In Python 3.7.1 the signatures of all context variables C APIs were changed to use PyObject pointers instead of PyContext, PyContextVar, and PyContextToken, e.g.:

```c
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

See bpo-34762 for more details.

New in version 3.7.

This section details the public C API for the contextvars module.

**PyContext**

The C structure used to represent a `contextvars.Context` object.

**PyContextVar**

The C structure used to represent a `contextvars.ContextVar` object.

**PyContextToken**

The C structure used to represent a `contextvars.Token` object.

**PyTypeObject** `PyContext_Type`
The type object representing the context type.

**PyTypeObject PyContextVar_Type**

The type object representing the context variable type.

**PyTypeObject PyContextToken_Type**

The type object representing the context variable token type.

Type-check macros:

```
int PyContext_CheckExact(PyObject *o)
    Return true if o is of type PyContext_Type. o must not be NULL. This function always succeeds.
```

```
int PyContextVar_CheckExact(PyObject *o)
    Return true if o is of type PyContextVar_Type. o must not be NULL. This function always succeeds.
```

```
int PyContextToken_CheckExact(PyObject *o)
    Return true if o is of type PyContextToken_Type. o must not be NULL. This function always succeeds.
```

Context object management functions:

```
PyObject *PyContext_New(void)
    Create a new empty context object. Returns NULL if an error has occurred.
```

```
PyObject *PyContext_Copy(PyObject *ctx)
    Create a shallow copy of the passed ctx context object. Returns NULL if an error has occurred.
```
PyObject *PyContext_CopyCurrent(void)

*Return value: New reference.*
Create a shallow copy of the current thread context. Returns NULL if an error has occurred.

int PyContext_Enter(PyObject *ctx)
Set ctx as the current context for the current thread. Returns 0 on success, and -1 on error.

int PyContext_Exit(PyObject *ctx)
Deactivate the ctx context and restore the previous context as the current context for the current thread. Returns 0 on success, and -1 on error.

int PyContext_ClearFreeList()
Clear the context variable free list. Return the total number of freed items. This function always succeeds.

Context variable functions:

PyObject *PyContextVar_New(const char *name, PyObject *def)

*Return value: New reference.*
Create a new ContextVar object. The name parameter is used for introspection and debug purposes. The def parameter may optionally specify the default value for the context variable. If an error has occurred, this function returns NULL.

int PyContextVar_Get(PyObject *var, PyObject *default_value, PyObject **value)
Get the value of a context variable. Returns -1 if an error has occurred during lookup, and 0 if no error occurred, whether or not a value was found.
If the context variable was found, value will be a pointer to it. If the context variable was not found, value will point to:

- `default_value`, if not NULL;
- the default value of `var`, if not NULL;
- NULL

If the value was found, the function will create a new reference to it.

```c
PyObject *PyContextVar_Set(PyObject *var, PyObject *value)
```

*Return value: New reference.*

Set the value of `var` to `value` in the current context. Returns a pointer to a `PyObject` object, or NULL if an error has occurred.

```c
int PyContextVar_Reset(PyObject *var, PyObject *token)
```

Reset the state of the `var` context variable to that it was in before `PyContextVar_Set()` that returned the `token` was called. This function returns 0 on success and -1 on error.
DateTime Objects

Various date and time objects are supplied by the `datetime` module. Before using any of these functions, the header file `datetime.h` must be included in your source (note that this is not included by `Python.h`), and the macro `PyDateTime_IMPORT` must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, `PyDateTimeAPI`, that is used by the following macros.

Macro for access to the UTC singleton:

```
PyObject* PyDateTime_TimeZone_UTC
```

Returns the time zone singleton representing UTC, the same object as `datetime.timezone.utc`. *New in version 3.7.*

Type-check macros:

```
int PyDate_Check(PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateType` or a subtype of `PyDateTime_DateType`. `ob` must not be `NULL`.

```
int PyDate_CheckExact(PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateType`. `ob` must not be `NULL`.

```
int PyDateTime_Check(PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateTimeType` or a subtype of `PyDateTime_DateTimeType`. `ob` must not be `NULL`. 
int **PyDateTime**_**_CheckExact**(PyObject *ob)
    Return true if ob is of type PyDateTime_DateTimeType. ob must not be NULL.

int **PyTime**_**_Check**(PyObject *ob)
    Return true if ob is of type PyDateTime_TimeType or a subtype of PyDateTime_TimeType. ob must not be NULL.

int **PyTime**_**_CheckExact**(PyObject *ob)
    Return true if ob is of type PyDateTime_TimeType. ob must not be NULL.

int **PyDelta**_**_Check**(PyObject *ob)
    Return true if ob is of type PyDateTime_DeltaType or a subtype of PyDateTime_DeltaType. ob must not be NULL.

int **PyDelta**_**_CheckExact**(PyObject *ob)
    Return true if ob is of type PyDateTime_DeltaType. ob must not be NULL.

int **PyTZInfo**_**_Check**(PyObject *ob)
    Return true if ob is of type PyDateTime_TZInfoType or a subtype of PyDateTime_TZInfoType. ob must not be NULL.

int **PyTZInfo**_**_CheckExact**(PyObject *ob)
    Return true if ob is of type PyDateTime_TZInfoType. ob must not be NULL.

Macros to create objects:

PyObject* **PyDate**_**_FromDate**(int year, int month, int day)
Return a `datetime.date` object with the specified year, month and day.

```
PyObject* PyDateTime_FromDateAndTime(int year, int month, int day, int hour, int minute, int second, int usecond)
```

*Return value: New reference.*

Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second and microsecond.

```
PyObject* PyTime_FromTime(int hour, int minute, int second, int usecond)
```

*Return value: New reference.*

Return a `datetime.time` object with the specified hour, minute, second and microsecond.

```
PyObject* PyDelta_FromDSU(int days, int seconds, int useconds)
```

*Return value: New reference.*

Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for `datetime.timedelta` objects.

```
PyObject*
PyTimeZone_FromOffset(PyDateTime_DeltaType* offset)
```

*Return value: New reference.*

Return a `datetime.timezone` object with an unnamed fixed offset represented by the `offset` argument.

*New in version 3.7.*

```
PyObject*
PyTimeZone_FromOffsetAndName(PyDateTime_DeltaType* offset, int timezone)
```

*Return value: New reference.*
PyUnicode* name)

Return a datetime.timezone object with a fixed offset represented by the offset argument and with tzname name.

New in version 3.7.

Macros to extract fields from date objects. The argument must be an instance of PyDateTime_Date, including subclasses (such as PyDateTime_DateTime). The argument must not be NULL, and the type is not checked:

int PyDateTime_GET_YEAR(PyDateTime_Date *o)
Return the year, as a positive int.

int PyDateTime_GET_MONTH(PyDateTime_Date *o)
Return the month, as an int from 1 through 12.

int PyDateTime_GET_DAY(PyDateTime_Date *o)
Return the day, as an int from 1 through 31.

Macros to extract fields from datetime objects. The argument must be an instance of PyDateTime_DateTime, including subclasses. The argument must not be NULL, and the type is not checked:

int PyDateTime_DATE_GET_HOUR(PyDateTime_DateTime *o)
Return the hour, as an int from 0 through 23.

int PyDateTime_DATE_GET_MINUTE(PyDateTime_DateTime *o)
Return the minute, as an int from 0 through 59.

int PyDateTime_DATE_GET_SECOND(PyDateTime_DateTime *o)
Return the second, as an int from 0 through 59.

```c
int PyDateTime_DATE_GET_MICROSECOND(PyDateTime_DateTime *o)

Return the microsecond, as an int from 0 through 999999.
```

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be `NULL`, and the type is not checked:

```c
int PyDateTime_TIME_GET_HOUR(PyDateTime_Time *o)

Return the hour, as an int from 0 through 23.
```

```c
int PyDateTime_TIME_GET_MINUTE(PyDateTime_Time *o)

Return the minute, as an int from 0 through 59.
```

```c
int PyDateTime_TIME_GET_SECOND(PyDateTime_Time *o)

Return the second, as an int from 0 through 59.
```

```c
int PyDateTime_TIME_GET_MICROSECOND(PyDateTime_Time *o)

Return the microsecond, as an int from 0 through 999999.
```

Macros to extract fields from time delta objects. The argument must be an instance of `PyDateTime_Delta`, including subclasses. The argument must not be `NULL`, and the type is not checked:

```c
int PyDateTime_DELTA_GET_DAYS(PyDateTime_Delta *o)

Return the number of days, as an int from -999999999 to 999999999.

*New in version 3.3.*
```

```c
int PyDateTime_DELTA_GET_SECONDS(PyDateTime_Delta *o)
```
Return the number of seconds, as an int from 0 through 86399.

_New in version 3.3._

```c
int PyDateTime_DELTA_GET_SECONDS(PyDateTime_Delta delta)
```

Return the number of microseconds, as an int from 0 through 999999.

_New in version 3.3._

Macros for the convenience of modules implementing the DB API:

```c
PyObject* PyDateTime_FromTimestamp(PyObject *args)
```

_Return value: New reference._

Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`.

```c
PyObject* PyDate_FromTimestamp(PyObject *args)
```

_Return value: New reference._

Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`.
Initialization, Finalization, and Threads
Before Python Initialization

In an application embedding Python, the \texttt{Py\_Initialize()} function must be called before using any other Python/C API functions; with the exception of a few functions and the \textit{global configuration variables}.

The following functions can be safely called before Python is initialized:

- **Configuration functions:**
  - \texttt{PyImport\_AppendInittab()}
  - \texttt{PyImport\_ExtendInittab()}
  - \texttt{PyInitFrozenExtensions()}
  - \texttt{PyMem\_SetAllocator()}
  - \texttt{PyMem\_SetupDebugHooks()}
  - \texttt{PyObject\_SetArenaAllocator()}
  - \texttt{Py\_SetPath()}
  - \texttt{Py\_SetProgramName()}
  - \texttt{Py\_SetPythonHome()}
  - \texttt{Py\_SetStandardStreamEncoding()}
  - \texttt{PySys\_AddWarnOption()}
  - \texttt{PySys\_AddXOption()}
  - \texttt{PySys\_ResetWarnOptions()}

- **Informative functions:**
  - \texttt{PyMem\_GetAllocator()}
  - \texttt{PyObject\_GetArenaAllocator()}
  - \texttt{Py\_GetBuildInfo()}
  - \texttt{Py\_GetCompiler()}
  - \texttt{Py\_GetCopyright()}
  - \texttt{Py\_GetPlatform()}
  - \texttt{Py\_GetVersion()}

- **Utilities:**
- `Py_DecodeLocale()`
- **Memory allocators:**
  - `PyMem_RawMalloc()`  
  - `PyMem_RawRealloc()`  
  - `PyMem_RawCalloc()`  
  - `PyMem_RawFree()`

**Note:** The following functions *should not be called* before `Py_Initialize()`: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` and `PyEval_InitThreads()`. 
Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets `Py_BytesWarningFlag` to 1 and `-bb` sets `Py_BytesWarningFlag` to 2.

**Py_BytesWarningFlag**

Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to 2.

Set by the `-b` option.

**Py_DebugFlag**

Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

**Py_DontWriteBytecodeFlag**

If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

**Py_FrozenFlag**

Suppress error messages when calculating the module search
path in `Py_GetPath()`.

Private flag used by `_freeze_importlib` and frozenmain programs.

**Py_HashRandomizationFlag**
Set to 1 if the `PYTHONHASHSEED` environment variable is set to a non-empty string.

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

**Py_IgnoreEnvironmentFlag**
Ignore all PYTHON* environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Set by the `-E` and `-I` options.

**Py_InspectFlag**
When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Set by the `-i` option and the `PYTHONINSPECT` environment variable.

**Py_InteractiveFlag**
Set by the `-i` option.

**Py_IsolatedFlag**
Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.
Set by the \texttt{-I} option.

\textit{New in version 3.4.}

**\texttt{Py\_LegacyWindowsFSEncodingFlag}**

If the flag is non-zero, use the \texttt{mbcs} encoding instead of the UTF-8 encoding for the filesystem encoding.

Set to \texttt{1} if the \texttt{PYTHONLEGACYWINDOWSFSENCODING} environment variable is set to a non-empty string.

See \texttt{PEP 529} for more details.

\textit{Availability: Windows.}

**\texttt{Py\_LegacyWindowsStdioFlag}**

If the flag is non-zero, use \texttt{io.FileIO} instead of \texttt{WindowsConsoleIO} for \texttt{sys} standard streams.

Set to \texttt{1} if the \texttt{PYTHONLEGACYWINDOWSSTDIO} environment variable is set to a non-empty string.

See \texttt{PEP 528} for more details.

\textit{Availability: Windows.}

**\texttt{Py\_NoSiteFlag}**

Disable the import of the module \texttt{site} and the site-dependent manipulations of \texttt{sys.path} that it entails. Also disable these manipulations if \texttt{site} is explicitly imported later (call \texttt{site.main()} if you want them to be triggered).

Set by the \texttt{-S} option.
**Py_NoUserSiteDirectory**
Don't add the user site-packages directory to sys.path.

Set by the -s and -I options, and the PYTHONNOUSERSITE environment variable.

**Py_OptimizeFlag**
Set by the -O option and the PYTHONOPTIMIZE environment variable.

**Py_QuietFlag**
Don't display the copyright and version messages even in interactive mode.

Set by the -q option.

*New in version 3.2.*

**Py_UnbufferedStdioFlag**
Force the stdout and stderr streams to be unbuffered.

Set by the -u option and the PYTHONUNBUFFERED environment variable.

**Py_VerboseFlag**
Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to 2, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the -v option and the PYTHONVERBOSE environment variable.
Initializing and finalizing the interpreter

void Py_Initialize()

Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see Before Python Initialization for the few exceptions.

This initializes the table of loaded modules (sys.modules), and creates the fundamental modules builtins, __main__ and sys. It also initializes the module search path (sys.path). It does not set sys.argv; use PySys_SetArgvEx() for that. This is a no-op when called for a second time (without calling Py_FinalizeEx() first). There is no return value; it is a fatal error if the initialization fails.

Note: On Windows, changes the console mode from 0_TEXT to 0_BINARY, which will also affect non-Python uses of the console using the C Runtime.

void Py_InitializeEx(int initsigs)

This function works like Py_Initialize() if initsigs is 1. If initsigs is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int Py_IsInitialized()

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After Py_FinalizeEx() is called, this returns false until Py_Initialize() is called again.

int Py_FinalizeEx()
Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first). Normally the return value is 0. If there were errors during finalization (flushing buffered data), -1 is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

**Bugs and caveats:** The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once.
New in version 3.6.

void **Py_Finalize**()

This is a backwards-compatible version of **Py_FinalizeEx()** that disregards the return value.
Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

This function should be called before **Py_Initialize()**, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in **str.encode()**.

It overrides **PYTHONIOENCODING** values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or errors may be NULL to use **PYTHONIOENCODING** and/or default values (depending on other settings).

Note that **sys.stderr** always uses the "backslashreplace" error handler, regardless of this (or any other) setting.

If **Py_FinalizeEx()** is called, this function will need to be called again in order to affect subsequent calls to **Py_Initialize()**.

Returns 0 if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

*New in version 3.4.*

void **Py_SetProgramName**(const wchar_t *name)

This function should be called before **Py_Initialize()** is called for the first time, if it is called at all. It tells the interpreter the value of the argv[0] argument to the main() function of the program (converted to wide characters). This is used by **Py_GetPath()**
and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is 'python'. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a wchar_* string.

wchar* **Py_GetProgramName()**

Return the program name set with `Py_SetProgramName()`, or the default. The returned string points into static storage; the caller should not modify its value.

wchar_t* **Py_GetPrefix()**

Return the prefix for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is '/usr/local/bin/python', the prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the prefix variable in the top-level Makefile and the --prefix argument to the configure script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

wchar_t* **Py_GetExecPrefix()**

Return the exec-prefix for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is
'usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the exec_prefix variable in the top-level Makefile and the --exec-prefix argument to the configure script at build time. The value is available to Python code as sys.exec_prefix. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the /usr/local/plat subtree while platform independent may be installed in /usr/local.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the mount or automount programs to share /usr/local between platforms while having /usr/local/plat be a different filesystem for each platform.
wchar_t* **Py_GetProgramFullPath()**

Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by **Py_SetProgramName()** above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as **sys.executable**.

wchar_t* **Py_GetPath()**

Return the default module search path; this is computed from the program name (set by **Py_SetProgramName()** above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is ': ' on Unix and Mac OS X, ';' on Windows. The returned string points into static storage; the caller should not modify its value. The list **sys.path** is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

void **Py_SetPath**(const wchar_t *)

Set the default module search path. If this function is called before **Py_Initialize()**, then **Py_GetPath()** won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is ': ' on Unix and Mac OS X, ';' on Windows.

This also causes **sys.executable** to be set only to the raw program name (see **Py_SetProgramName()**)) and for **sys.prefix** and **sys.exec_prefix** to be empty. It is up to the
caller to modify these if required after calling Py_Initinitialize().

Use Py_DecodeLocale() to decode a bytes string to get a wchar_* string.

The path argument is copied internally, so the caller may free it after the call completes.

const char* Py_GetVersion()
Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[G
```

The first word (up to the first space character) is the current Python version; the first three characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.version.

const char* Py_GetPlatform()
Return the platform identifier for the current platform. On Unix, this is formed from the “official” name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is 'sunos5'. On Mac OS X, it is 'darwin'. On Windows, it is 'win'. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.platform.

const char* Py_GetCopyright()
Return the official copyright string for the current Python version,
for example

'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char* `Py_GetCompiler()`
Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char* `Py_GetBuildInfo()`
Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug 1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void `PySys_SetArgvEx`(int argc, wchar_t **argv, int updatepath)
Set `sys.argv` based on `argc` and `argv`. These parameters are similar to those passed to the program’s main() function with the difference that the first entry should refer to the script file to be
executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in `argv` can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using `Py_FatalError()`.

If `updatepath` is zero, this is all the function does. If `updatepath` is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if `argc` is 0 or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (".").

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

**Note:** It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass 0 as `updatepath`, and update `sys.path` themselves if desired. See CVE-2008-5983.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using:

```python
PyRun_SimpleString("import sys; sys.path.pop(0)\n")
```

*New in version 3.1.3.*
void **PySys_SetArgv(int argc, wchar_t **argv)
    This function works like PySys_SetArgvEx() with updatepath set to 1 unless the python interpreter was started with the -I.

    Use Py_DecodeLocale() to decode a bytes string to get a wchar_* string.

    Changed in version 3.4: The updatepath value depends on -I.

void Py_SetPythonHome(const wchar_t *home)
    Set the default “home” directory, that is, the location of the standard Python libraries. See PYTHONHOME for the meaning of the argument string.

    The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.

    Use Py_DecodeLocale() to decode a bytes string to get a wchar_* string.

w_char** Py_GetPythonHome()
    Return the default “home”, that is, the value set by a previous call to Py_SetPythonHome(), or the value of the PYTHONHOME environment variable if it is set.
Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called `PyThreadState`. There's also one global variable pointing to the current `PyThreadState`: it can be retrieved using `PyThreadState_Get()`.

Releasing the GIL from extension code

Most extension code manipulating the GIL has the following simple structure:

Save the thread state in a local variable.  
Release the global interpreter lock.
... Do some blocking I/O operation ...  
Reacquire the global interpreter lock.  
Restore the thread state from the local variable.

This is so common that a pair of macros exists to simplify it:

```python
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block.

The block above expands to the following code:

```python
PyThreadState *__save;
__save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(__save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

**Note:** Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don’t need access to Python objects,
such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don’t hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The `PyGILState_Ensure()` and `PyGILState_Release()` functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```c
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */
```
Note that the PyGILState_*() functions assume there is only one global interpreter (created automatically by Py_Initialize()). Python supports the creation of additional interpreters (using Py_NewInterpreter()), but mixing multiple interpreters and the PyGILState_*() API is unsupported.

Another important thing to note about threads is their behaviour in the face of the C fork() call. On most systems with fork(), after a process forks only the thread that issued the fork will exist. That also means any locks held by other threads will never be released. Python solves this for os.fork() by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any Lock Objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as pthread_atfork() would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling fork() directly rather than through os.fork() (and returning to or calling into Python) may result in a deadlock by one of Python’s internal locks being held by a thread that is defunct after the fork. PyOS_AfterFork_Child() tries to reset the necessary locks, but is not always able to.

High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:
**PyInterpreterState**

This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

**PyThreadState**

This data structure represents the state of a single thread. The only public data member is `PyInterpreterState *interp`, which points to this thread's interpreter state.

```c
void PyEval_InitThreads()
```

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time.

*Changed in version 3.7:* This function is now called by `Py_Initialize()`, so you don’t have to call it yourself anymore.

*Changed in version 3.2:* This function cannot be called before `Py_Initialize()` anymore.

```c
int PyEval_ThreadsInitialized()
```

Returns a non-zero value if `PyEval_InitThreads()` has been
called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

*Changed in version 3.7:* The GIL is now initialized by `Py_Initialize()`.

**PyThreadState** `PyEval_SaveThread()`  
Release the global interpreter lock (if it has been created and thread support is enabled) and reset the thread state to `NULL`, returning the previous thread state (which is not `NULL`). If the lock has been created, the current thread must have acquired it.

```c
void PyEval_RestoreThread(PyThreadState *tstate)
```

Acquire the global interpreter lock (if it has been created and thread support is enabled) and set the thread state to `tstate`, which must not be `NULL`. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

**PyThreadState** `PyThreadState_Get()`  
Return the current thread state. The global interpreter lock must be held. When the current thread state is `NULL`, this issues a fatal error (so that the caller needn’t check for `NULL`).

**PyThreadState** `PyThreadState_Swap(PyThreadState *tstate)`  
Swap the current thread state with the thread state given by the argument `tstate`, which may be `NULL`. The global interpreter lock must be held and is not released.

```c
void PyEval_ReInitThreads()
```

This function is called from `PyOS_AfterFork_Child()` to ensure that newly created child processes don’t hold locks referring to threads which are not running in the child process.
The following functions use thread-local storage, and are not compatible with sub-interpreters:

**PyGILState_STATE PyGILState_Ensure()**

Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to **PyGILState_Release()**. In general, other thread-related APIs may be used between **PyGILState_Ensure()** and **PyGILState_Release()** calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the **Py_BEGIN_ALLOW_THREADS** and **Py_END_ALLOW_THREADS** macros is acceptable.

The return value is an opaque “handle” to the thread state when **PyGILState_Ensure()** was called, and must be passed to **PyGILState_Release()** to ensure Python is left in the same state. Even though recursive calls are allowed, these handles cannot be shared - each unique call to **PyGILState_Ensure()** must save the handle for its call to **PyGILState_Release()**.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

**void PyGILState_Release(PyGILState_STATE)**

Release any resources previously acquired. After this call, Python’s state will be the same as it was prior to the corresponding **PyGILState_Ensure()** call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to **PyGILState_Ensure()** must be matched by a call
to \texttt{PyGILState\_Release()} on the same thread.

\textbf{PyThreadState* \texttt{PyGILState\_GetThisThreadState}()} \\
Get the current thread state for this thread. May return \texttt{NULL} if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

\textbf{int \texttt{PyGILState\_Check}()} \\
Return 1 if the current thread is holding the GIL and 0 otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return 1. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

\textit{New in version 3.4.}

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

\textbf{Py\_BEGIN\_ALLOW\_THREADS} \\
This macro expands to \texttt{\{ PyThreadState *\_save; \_save = PyEval\_SaveThread(); }\}. Note that it contains an opening brace; it must be matched with a following \texttt{Py\_END\_ALLOW\_THREADS} macro. See above for further discussion of this macro.

\textbf{Py\_END\_ALLOW\_THREADS} \\
This macro expands to \texttt{PyEval\_RestoreThread(\_save); \}}.
Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro.

**Py_BLOCK_THREADS**

This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace.

**Py_UNBLOCK_THREADS**

This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to `Py_BEGIN_ALLOW_THREADS` without the opening brace and variable declaration.

Low-level API

All of the following functions must be called after `Py_Initialize()`.

*Changed in version 3.7:* `Py_Initialize()` now initializes the GIL.

`PyInterpreterState* PyInterpreterState_New()`  
Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

`void PyInterpreterState_Clear(PyInterpreterState *interp)`  
Reset all information in an interpreter state object. The global interpreter lock must be held.

`void PyInterpreterState_Delete(PyInterpreterState *interp)`  
Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a
previous call to `PyInterpreterState_Clear()`.

**PyThreadState** `PyThreadState_New(PyInterpreterState *interp)`

Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

**void PyThreadState_Clear(PyThreadState *tstate)**

Reset all information in a thread state object. The global interpreter lock must be held.

**void PyThreadState_Delete(PyThreadState *tstate)**

Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to `PyThreadState_Clear()`.

**PY_INT64_T PyInterpreterState_GetID(PyInterpreterState *interp)**

Return the interpreter’s unique ID. If there was any error in doing so then -1 is returned and an error is set.

*New in version 3.7.*

**PyObject* PyThreadState_GetDict()**

*Return value: Borrowed reference.*

Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns `NULL`, no exception has been raised and the caller should assume no current thread state is available.

**int PyThreadState_SetAsyncExc(unsigned long id,**
Asynchronously raise an exception in a thread. The id argument is the thread id of the target thread; exc is the exception object to be raised. This function does not steal any references to exc. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If exc is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions.

*Changed in version 3.7:* The type of the id parameter changed from long to unsigned long.

void **PyEval_AcquireThread**(PyThreadState *tstate)

Acquire the global interpreter lock and set the current thread state to tstate, which should not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

**PyEval_RestoreThread()** is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(PyThreadState *tstate)

Reset the current thread state to NULL and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The tstate argument, which must not be NULL, is only used to check that it represents the current thread state — if it isn’t, a fatal error is reported.

**PyEval_SaveThread()** is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_AcquireLock**()
Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

*Deprecated since version 3.2:* This function does not update the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

```c
void PyEval_ReleaseLock()
```

Release the global interpreter lock. The lock must have been created earlier.

*Deprecated since version 3.2:* This function does not update the current thread state. Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.
Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that. You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions:

*PyThreadState* `Py_NewInterpreter()`

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, `NULL` is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn’t be a current thread
Extension modules are shared between (sub-)interpreters as follows: the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's init function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling Py_FinalizeEx() and Py_Initialize(); in that case, the extension's initmodule function is called again.

**void Py_EndInterpreter(PyThreadState *tstate)**

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is NULL. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) Py_FinalizeEx() will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

**Bugs and caveats**

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like os.close() they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the
extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter’s dictionary of loaded modules.

Also note that combining this functionality with PyGILState_*( ) APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don’t switch sub-interpreters between a pair of matching PyGILState_Ensure( ) and PyGILState_Release( ) calls. Furthermore, extensions (such as ctypes) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.
Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

```c
int Py_AddPendingCall(int (*func)(void *), void *arg)
```

Schedule a function to be called from the main interpreter thread. On success, `0` is returned and `func` is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, `func` will be eventually called from the main interpreter thread with the argument `arg`. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a bytecode boundary;
- with the main thread holding the global interpreter lock (`func` can therefore use the full C API).

`func` must return `0` on success, or `-1` on failure with an exception set. `func` won’t be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn’t need a current thread state to run, and it doesn’t need the global interpreter lock.

**Warning:** This is a low-level function, only useful for very special cases. There is no guarantee that `func` will be called as quick as possible. If the main thread is busy executing a system call, `func` won’t be called before the system call returns. This
function is generally not suitable for calling Python code from arbitrary C threads. Instead, use the PyGILState API.

New in version 3.1.
Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

```c
int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)
```

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as `obj`, `frame` is the frame object to which the event pertains, `what` is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and `arg` depends on the value of `what`:

<table>
<thead>
<tr>
<th>Value of <code>what</code></th>
<th>Meaning of <code>arg</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTrace_CALL</td>
<td>Always <code>Py_None</code>.</td>
</tr>
<tr>
<td>PyTrace_EXCEPTION</td>
<td>Exception information as returned by <code>sys.exc_info()</code></td>
</tr>
<tr>
<td>PyTrace_LINE</td>
<td>Always <code>Py_None</code>.</td>
</tr>
<tr>
<td>PyTrace_RETURN</td>
<td>Value being returned to the caller, or NULL if caused by an exception.</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>PyTrace_C_CALL</td>
<td>Function object being called.</td>
</tr>
<tr>
<td>PyTrace_C_EXCEPTION</td>
<td>Function object being called.</td>
</tr>
<tr>
<td>PyTrace_C_RETURN</td>
<td>Function object being called.</td>
</tr>
<tr>
<td>PyTrace_OPCODE</td>
<td>Always Py_None.</td>
</tr>
</tbody>
</table>

int **PyTrace_CALL**

The value of the what parameter to a Py_tracefunc function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the what parameter to a Py_tracefunc function when an exception has been raised. The callback function is called with this value for what when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the what parameter to a Py_tracefunc function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting f_trace_lines to 0 on that frame.

int **PyTrace_RETURN**
The value for the what parameter to Py_tracefunc functions when a call is about to return.

int PyTrace_C_CALL
The value for the what parameter to Py_tracefunc functions when a C function is about to be called.

int PyTrace_C_EXCEPTION
The value for the what parameter to Py_tracefunc functions when a C function has raised an exception.

int PyTrace_C_RETURN
The value for the what parameter to Py_tracefunc functions when a C function has returned.

int PyTrace_OPCODE
The value for the what parameter to Py_tracefunc functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting f_trace_opcodes to 1 on the frame.

void PyEval_SetProfile(Py_tracefunc func, PyObject *obj)
Set the profiler function to func. The obj parameter is passed to the function as its first parameter, and may be any Python object, or NULL. If the profile function needs to maintain state, using a different value for obj for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except PyTrace_LINE PyTrace_OPCODE and PyTrace_EXCEPTION.

void PyEval_SetTrace(Py_tracefunc func, PyObject *obj)
Set the tracing function to func. This is similar to
PyEval_SetProfile(), except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using PyEval_SetTrace() will not receive PyTrace_C_CALL, PyTrace_C_EXCEPTION or PyTrace_C_RETURN as a value for the what parameter.
Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

**PyInterpreterState**

*PyInterpreterState* Head()

Return the interpreter state object at the head of the list of all such objects.

*PyInterpreterState*

PyInterpreterState Next(PyInterpreterState *interp)

Return the next interpreter state object after interp from the list of all such objects.

*PyThreadState*

PyInterpreterState_ThreadHead(PyInterpreterState *interp)

Return the pointer to the first PyThreadState object in the list of threads associated with the interpreter interp.

*PyThreadState*

PyThreadState Next(PyThreadState *tstate)

Return the next thread state object after tstate from the list of all such objects belonging to the same PyInterpreterState object.
Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (threading.local). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a void* value per thread.

The GIL does not need to be held when calling these functions; they supply their own locking.

Note that Python.h does not include the declaration of the TLS APIs, you need to include pythread.h to use thread-local storage.

**Note:** None of these API functions handle memory management on behalf of the void* values. You need to allocate and deallocate them yourself. If the void* values happen to be PyObject*, these functions don’t do refcount operations on them either.

Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type Py_tss_t instead of int to represent thread keys.

*New in version 3.7.*

**See also:** “A New C-API for Thread-Local Storage in CPython” (PEP 539)
**Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When `Py_LIMITED_API` is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

**Py_tss_NEEDS_INIT**

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won’t be defined with `Py_LIMITED_API`.

**Dynamic Allocation**

Dynamic allocation of the `Py_tss_t`, required in extension modules built with `Py_LIMITED_API`, where static allocation of this type is not possible due to its implementation being opaque at build time.

**Py_tss_t** `PyThread_tss_alloc()`

Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

void `PyThread_tss_free(Py_tss_t *key)`

Free the given key allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the key argument is `NULL`.

**Note:** A freed key becomes a dangling pointer, you should reset the key to `NULL`. 
Methods

The parameter *key* of these functions must not be *NULL*. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

```c
int PyThread_tss_is_created(Py_tss_t *key)
```

Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

```c
int PyThread_tss_create(Py_tss_t *key)
```

Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

```c
void PyThread_tss_delete(Py_tss_t *key)
```

Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

```c
int PyThread_tss_set(Py_tss_t *key, void *value)
```

Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

```c
void* PyThread_tss_get(Py_tss_t *key)
```
Return the void* value associated with a TSS key in the current thread. This returns NULL if no value is associated with the key in the current thread.

**Thread Local Storage (TLS) API**

_Deprecated since version 3.7:_ This API is superseded by Thread Specific Storage (TSS) API.

**Note:** This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to int. On such platforms, `PyThread_create_key()` will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

```c
int PyThread_create_key()

void PyThread_delete_key(int key)

int PyThread_set_key_value(int key, void *value)

void* PyThread_get_key_value(int key)

void PyThread_delete_key_value(int key)

void PyThread_ReInitTLS()
```
Memory Management
Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library:
malloc(), calloc(), realloc() and free(). This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```c
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to inform the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or
other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

See also: The PYTHONMALLOC environment variable can be used to configure the memory allocators used by Python.

The PYTHONMALLOCSTATS environment variable can be used to print statistics of the pymalloc memory allocator every time a new pymalloc object arena is created, and on shutdown.
Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the GIL does not need to be held.

The default raw memory allocator uses the following functions: malloc(), calloc(), realloc() and free(); call malloc(1) (or calloc(1, 1)) when requesting zero bytes.

New in version 3.4.

void* PyMem_RawMalloc(size_t n)

Allocates n bytes and returns a pointer of type void* to the allocated memory, or NULL if the request fails.

Requesting zero bytes returns a distinct non-NULL pointer if possible, as if PyMem_RawMalloc(1) had been called instead. The memory will not have been initialized in any way.

void* PyMem_RawCalloc(size_t nelem, size_t elsize)

Allocates nelem elements each whose size in bytes is elsize and returns a pointer of type void* to the allocated memory, or NULL if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-NULL pointer if possible, as if PyMem_RawCalloc(1, 1) had been called instead.

New in version 3.5.

void* PyMem_RawRealloc(void *p, size_t n)

Resizes the memory block pointed to by p to n bytes. The contents
will be unchanged to the minimum of the old and the new sizes.

If $p$ is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if $n$ is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless $p$ is `NULL`, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns `NULL` and $p$ remains a valid pointer to the previous memory area.

```c
void PyMem_RawFree(void *p)
```

Frees the memory block pointed to by $p$, which must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If $p$ is `NULL`, no operation is performed.
Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The default memory allocator uses the pymalloc memory allocator.

**Warning:** The **GIL** must be held when using these functions.

*Changed in version 3.6:* The default allocator is now pymalloc instead of system `malloc()`.

```c
void* PyMem_Malloc(size_t n)
    Allocates n bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

    Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if PyMem_Malloc(1) had been called instead. The memory will not have been initialized in any way.

```c
void* PyMem_Calloc(size_t nelem, size_t elsize)
    Allocates nelem elements each whose size in bytes is elsize and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

    Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if PyMem_Calloc(1, 1) had been called instead.

*New in version 3.5.*
void* PyMem_Realloc(void *p, size_t n)

Resizes the memory block pointed to by p to n bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If p is NULL, the call is equivalent to PyMem_Malloc(n); else if n is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL.

Unless p is NULL, it must have been returned by a previous call to PyMem_Malloc(), PyMem_Realloc() or PyMem_Calloc().

If the request fails, PyMem_Realloc() returns NULL and p remains a valid pointer to the previous memory area.

void PyMem_Free(void *p)

Frees the memory block pointed to by p, which must have been returned by a previous call to PyMem_Malloc(), PyMem_Realloc() or PyMem_Calloc(). Otherwise, or if PyMem_Free(p) has been called before, undefined behavior occurs.

If p is NULL, no operation is performed.

The following type-oriented macros are provided for convenience. Note that TYPE refers to any C type.

TYPE* PyMem_New(TYPE, size_t n)

Same as PyMem_Malloc(), but allocates (n * sizeof(TYPE)) bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.

TYPE* PyMem_Resize(void *p, TYPE, size_t n)

Same as PyMem_Realloc(), but the memory block is resized to
\( (n \times \text{sizeof}(\text{TYPE})) \) bytes. Returns a pointer cast to \text{TYPE}*. On return, \( p \) will be a pointer to the new memory area, or \text{NULL} in the event of failure.

This is a C preprocessor macro; \( p \) is always reassigned. Save the original value of \( p \) to avoid losing memory when handling errors.

```c
void PyMem_Del(void *p)
    Same as PyMem_Free().
```

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`
Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The **default object allocator** uses the **pymalloc memory allocator**.

**Warning:** The **GIL** must be held when using these functions.

```c
void* PyObject_Malloc(size_t n)
```

Allocates `n` bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

```c
void* PyObject_Calloc(size_t nelem, size_t elsize)
```

Allocates `nelem` elements each whose size in bytes is `elsize` and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

*New in version 3.5.*

```c
void* PyObject_Realloc(void* p, size_t n)
```

Resizes the memory block pointed to by `p` to `n` bytes. The contents will be unchanged to the minimum of the old and the new sizes.
If $p$ is $NULL$, the call is equivalent to $\text{PyObject_Malloc}(n)$; else if $n$ is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-$NULL$.

Unless $p$ is $NULL$, it must have been returned by a previous call to $\text{PyObject_Malloc()}$, $\text{PyObject_Realloc()}$ or $\text{PyObject_Calloc()}$.

If the request fails, $\text{PyObject_Realloc()}$ returns $NULL$ and $p$ remains a valid pointer to the previous memory area.

**void** $\text{PyObject_Free}(\text{void *}p)$

Frees the memory block pointed to by $p$, which must have been returned by a previous call to $\text{PyObject_Malloc()}$, $\text{PyObject_Realloc()}$ or $\text{PyObject_Calloc()}$. Otherwise, or if $\text{PyObject_Free}(p)$ has been called before, undefined behavior occurs.

If $p$ is $NULL$, no operation is performed.
# Default Memory Allocators

Default memory allocators:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Name</th>
<th>PyMem_RawMalloc</th>
<th>PyMem_Malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release build</td>
<td>&quot;pymalloc&quot;</td>
<td>malloc</td>
<td>pymalloc</td>
</tr>
<tr>
<td>Debug build</td>
<td>&quot;pymalloc_debug&quot;</td>
<td>malloc + debug</td>
<td>pymalloc_debug</td>
</tr>
<tr>
<td>Release build, without pymalloc</td>
<td>&quot;malloc&quot;</td>
<td>malloc</td>
<td>malloc</td>
</tr>
<tr>
<td>Debug build, without pymalloc</td>
<td>&quot;malloc_debug&quot;</td>
<td>malloc + debug</td>
<td>malloc_debug</td>
</tr>
</tbody>
</table>

Legend:

- **Name**: value for PYTHONMALLOC environment variable
- **malloc**: system allocators from the standard C library, C functions: malloc(), calloc(), realloc() and free()
- **pymalloc**: pymalloc memory allocator
- **"+ debug"**: with debug hooks installed by PyMem_SetupDebugHooks()
Customize Memory Allocators

New in version 3.4.

**PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has four fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *ctx</td>
<td>user context passed as first argument</td>
</tr>
<tr>
<td>void* malloc(void *ctx, size_t size)</td>
<td>allocate a memory block</td>
</tr>
<tr>
<td>void* calloc(void *ctx, size_t nelem, size_t elsize)</td>
<td>allocate a memory block initialized with zeros</td>
</tr>
<tr>
<td>void* realloc(void *ctx, void *ptr, size_t new_size)</td>
<td>allocate or resize a memory block</td>
</tr>
<tr>
<td>void free(void *ctx, void *ptr)</td>
<td>free a memory block</td>
</tr>
</tbody>
</table>

Changed in version 3.5: The PyMemAllocator structure was renamed to PyMemAllocatorEx and a new calloc field was added.

**PyMemAllocatorDomain**

Enum used to identify an allocator domain. Domains:

**PYMEM_DOMAIN_RAW**

Functions:

- PyMem_RawMalloc()
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`
- `PyMem_RawFree()`

**PYMEM_DOMAIN_MEM**

Functions:

- `PyMem_Malloc()`
- `PyMem_Realloc()`
- `PyMem_Calloc()`
- `PyMem_Free()`

**PYMEM_DOMAIN_OBJ**

Functions:

- `PyObject_Malloc()`
- `PyObject_Realloc()`
- `PyObject_Calloc()`
- `PyObject_Free()`

```c
void PyMem_GetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)
{
    Get the memory block allocator of the specified domain.
}

void PyMem_SetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)
{
    Set the memory block allocator of the specified domain.

    The new allocator must return a distinct non-NULL pointer when requesting zero bytes.

    For the **PYMEM_DOMAIN_RAW** domain, the allocator must be thread-safe: the **GIL** is not held when the allocator is called.
```
If the new allocator is not a hook (does not call the previous allocator), the `PyMem_SetupDebugHooks()` function must be called to reinstall the debug hooks on top on the new allocator.

```c
void PyMem_SetupDebugHooks(void)
```

Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte \(0xCB\), freed memory is filled with the byte \(0xDB\).

Runtime checks:

- Detect API violations, ex: `PyObject_Free()` called on a buffer allocated by `PyMem_Malloc()`
- Detect write before the start of the buffer (buffer underflow)
- Detect write after the end of the buffer (buffer overflow)
- Check that the GIL is held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

These hooks are installed by default if Python is compiled in debug mode. The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode.

*Changed in version 3.6:* This function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was
allocated. The debug hooks now also check if the GIL is held when functions of `PYMEM_DOMAIN_OBJ` and `PYMEM_DOMAIN_MEM` domains are called.
The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called “arenas” with a fixed size of 256 KiB. It falls back to `PyMem_RawMalloc()` and `PyMem_RawRealloc()` for allocations larger than 512 bytes.

*pymalloc* is the default allocator of the PYMEM_DOMAIN_MEM (ex: `PyMem_Malloc()`) and PYMEM_DOMAIN_OBJ (ex: `PyObject_Malloc()`) domains.

The arena allocator uses the following functions:

- VirtualAlloc() and VirtualFree() on Windows,
- mmap() and munmap() if available,
- malloc() and free() otherwise.

Customize pymalloc Arena Allocator

*New in version 3.4.*

**PyObjectArenaAllocator**

Structure used to describe an arena allocator. The structure has three fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *ctx</td>
<td>user context passed as first argument</td>
</tr>
<tr>
<td>void* alloc(void *ctx, size_t size)</td>
<td>allocate an arena of size bytes</td>
</tr>
</tbody>
</table>
free an arena

void free(void *ctx,
    size_t size,
    void *ptr)  

PyObject_GetArenaAllocator
Get the arena allocator.

PyObject_SetArenaAllocator
Set the arena allocator.
tracemalloc C API

New in version 3.7.
Examples

Here is the example from section Overview, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```c
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); // for I/O

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```c
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as fatal because it mixes two different allocators operating on different heaps.
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
free(buf2); /* Right -- allocated via malloc() */
free(buf1); /* Fatal -- should be PyMem_Del() */

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New()`, `PyObject_NewVar()` and `PyObject_Del()`.

These will be explained in the next chapter on defining and implementing new object types in C.
Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

- Allocating Objects on the Heap
- Common Object Structures
- Type Objects
- Number Object Structures
- Mapping Object Structures
- Sequence Object Structures
- Buffer Object Structures
- Async Object Structures
- Supporting Cyclic Garbage Collection
Allocating Objects on the Heap

PyObject* _PyObject_New(PyTypeObject *type)

*Return value: New reference.*

PyVarObject* _PyObject_NewVar(PyTypeObject *type, Py_ssize_t size)

*Return value: New reference.*

PyObject* PyObject_Init(PyObject *op, PyTypeObject *type)

*Return value: Borrowed reference.*
Initialize a newly-allocated object op with its type and initial reference. Returns the initialized object. If type indicates that the object participates in the cyclic garbage detector, it is added to the detector’s set of observed objects. Other fields of the object are not affected.

PyVarObject* PyObject_InitVar(PyVarObject *op, PyTypeObject *type, Py_ssize_t size)

*Return value: Borrowed reference.*
This does everything PyObject_Init() does, and also initializes the length information for a variable-size object.

TYPE* PyObject_New(TYPE, PyTypeObject *type)

*Return value: New reference.*
Allocate a new Python object using the C structure type TYPE and the Python type object type. Fields not defined by the Python object header are not initialized; the object’s reference count will be one. The size of the memory allocation is determined from the tp_basicsize field of the type object.

TYPE* PyObject_NewVar(TYPE, PyTypeObject *type,
Py_ssize_t size)

Allocate a new Python object using the C structure type TYPE and the Python type object type. Fields not defined by the Python object header are not initialized. The allocated memory allows for the TYPE structure plus size fields of the size given by the tp_itemsize field of type. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

void PyObject_Del(PyObject *op)

Releases memory allocated to an object using PyObject_New() or PyObject_NewVar(). This is normally called from the tp_dealloc handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

PyObject _Py_NoneStruct

Object which is visible in Python as None. This should only be accessed using the Py_None macro, which evaluates to a pointer to this object.

See also:

PyModule_Create()
To allocate and create extension modules.
Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object’s representation in memory. These are represented by the `PyObject` and `PyVarObject` types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

**PyObject**

All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal “release” build, it contains only the object’s reference count and a pointer to the corresponding type object. Nothing is actually declared to be a `PyObject`, but every pointer to a Python object can be cast to a `PyObject*`. Access to the members must be done by using the macros `Py_REFCNT` and `Py_TYPE`.

**PyVarObject**

This is an extension of `PyObject` that adds the `ob_size` field. This is only used for objects that have some notion of `length`. This type does not often appear in the Python/C API. Access to the members must be done by using the macros `Py_REFCNT`, `Py_TYPE`, and `Py_SIZE`.

**PyObject_HEAD**

This is a macro used when declaring new types which represent
objects without a varying length. The PyObject_HEAD macro expands to:

```c
PyObject ob_base;
```

See documentation of `PyObject` above.

**PyObject_VAR_HEAD**

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```c
PyVarObject ob_base;
```

See documentation of `PyVarObject` above.

**Py_TYPE(o)**

This macro is used to access the `ob_type` member of a Python object. It expands to:

```c
(((PyObject*)o)->ob_type)
```

**Py_REFCNT(o)**

This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```c
(((PyObject*)o)->ob_refcnt)
```

**Py_SIZE(o)**

This macro is used to access the `ob_size` member of a Python object. It expands to:

```c
(((PyVarObject*)o)->ob_size)```
PyObject\_HEAD\_INIT(type)

This is a macro which expands to initialization values for a newPyObject type. This macro expands to:

```
PyObject\_EXTRA\_INIT
1, type,
```

PyVarObject\_HEAD\_INIT(type, size)

This is a macro which expands to initialization values for a newPyVarObject type, including the ob\_size field. This macro expands to:

```
PyObject\_EXTRA\_INIT
1, type, size,
```

PyCFunction

Type of the functions used to implement most Python callables inC. Functions of this type take two PyObject\* parameters andreturn one such value. If the return value is NULL, an exceptionshall have been set. If not NULL, the return value is interpreted asthe return value of the function as exposed in Python. The functionmust return a new reference.

PyCFunctionWithKeywords

Type of the functions used to implement Python callables in C thattake keyword arguments: they take three PyObject\* parameters andreturn one such value. See PyCFunction above for the meaning of the return value.

PyMethodDef

Structure used to describe a method of an extension type. Thisstructure has four fields:
<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_name</td>
<td>const char *</td>
<td>name of the method</td>
</tr>
<tr>
<td>ml_meth</td>
<td>PyCFunction</td>
<td>pointer to the C implementation</td>
</tr>
<tr>
<td>ml_flags</td>
<td>int</td>
<td>flag bits indicating how the call should be constructed</td>
</tr>
<tr>
<td>ml_doc</td>
<td>const char *</td>
<td>points to the contents of the docstring</td>
</tr>
</tbody>
</table>

The `ml_meth` is a C function pointer. The functions may be of different types, but they always return `PyObject*`. If the function is not of the `PyCFunction`, the compiler will require a cast in the method table. Even though `PyCFunction` defines the first parameter as `PyObject*`, it is common that the method implementation uses the specific C type of the `self` object.

The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention. Of the calling convention flags, only `METH_VARARGS` and `METH_KEYWORDS` can be combined. Any of the calling convention flags can be combined with a binding flag.

**METH_VARARGS**

This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two `PyObject*` values. The first one is the `self` object for methods; for module functions, it is the module object. The second parameter (often called `args`) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.
**METH_KEYWORDS**

Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: `self`, `args`, and a dictionary of all the keyword arguments. The flag must be combined with `METH_VARARGS`, and the parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

**METH_NOARGS**

Methods without parameters don’t need to check whether arguments are given if they are listed with the `METH_NOARGS` flag. They need to be of type `PyCFunction`. The first parameter is typically named `self` and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

**METH_O**

Methods with a single object argument can be listed with the `METH_O` flag, instead of invoking `PyArg_ParseTuple()` with a "0" argument. They have the type `PyCFunction`, with the `self` parameter, and a `PyObject*` parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

**METH_CLASS**

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create `class methods`, similar to what is created when using the `classmethod()` built-in function.
**METH_STATIC**

The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

**METH_COEXIST**

The method will be loaded in place of existing definitions. Without `METH_COEXIST`, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a `sq_contains` slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding `PyCFunction` with the same name. With the flag defined, the `PyCFunction` will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to `PyCFunctions` are optimized more than wrapper object calls.

**PyMemberDef**

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td><code>const char *</code></td>
<td>name of the member</td>
</tr>
<tr>
<td>type</td>
<td><code>int</code></td>
<td>the type of the member in the C struct</td>
</tr>
<tr>
<td>offset</td>
<td><code>Py_ssize_t</code></td>
<td>the offset in bytes that the member is located on the type’s object struct</td>
</tr>
</tbody>
</table>
flags | int | flag bits indicating if the field should be read-only or writable
--- | --- | ---
doc | const char * | points to the contents of the docstring

Type can be one of many T_ macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_SHORT</td>
<td>short</td>
</tr>
<tr>
<td>T_INT</td>
<td>int</td>
</tr>
<tr>
<td>T_LONG</td>
<td>long</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>T_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>T_STRING</td>
<td>const char *</td>
</tr>
<tr>
<td>T_OBJECT</td>
<td>PyObject *</td>
</tr>
<tr>
<td>T_OBJECT_EX</td>
<td>PyObject *</td>
</tr>
<tr>
<td>T_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>T_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>T_UINT</td>
<td>unsigned int</td>
</tr>
<tr>
<td>T_USHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>T_ULONGLONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>T_BOOL</td>
<td>char</td>
</tr>
<tr>
<td>T_LONGLONG</td>
<td>long long</td>
</tr>
<tr>
<td>Field</td>
<td>C Type</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>name</td>
<td>const char *</td>
</tr>
<tr>
<td>get</td>
<td>getter</td>
</tr>
<tr>
<td>set</td>
<td>setter</td>
</tr>
<tr>
<td>doc</td>
<td>const char *</td>
</tr>
<tr>
<td>closure</td>
<td>void *</td>
</tr>
</tbody>
</table>

PyGetSetDef

Structure to define property-like access for a type. See also description of the PyTypeObject.tp_getset slot.

The get function takes one PyObject* parameter (the instance) and a function pointer (the associated closure):
**typedef** PyObject *((*getter)(PyObject *, void *)));

It should return a new reference on success or *NULL* with a set exception on failure.

**set** functions take two *PyObject* parameters (the instance and the value to be set) and a function pointer (the associated closure):

```c
**typedef** int (*setter)(PyObject *, PyObject *, void *
```

In case the attribute should be deleted the second parameter is *NULL*. Should return 0 on success or -1 with a set exception on failure.
Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the PyTypeObject structure. Type objects can be handled using any of the PyObject_*() or PyType_*() functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

Typedefs: unaryfunc, binaryfunc, ternaryfunc, inquiry, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, freefunc, printfunc, getattrfunc, getattrofunc, setattrfunc, setattrofunc, reprfunc, hashfunc

The structure definition for PyTypeObject can be found in Include/object.h. For convenience of reference, this repeats the definition found there:

```c
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For all */
    /* Methods to implement standard operations */
```
destructor tp_dealloc;
printfunc tp_print;
getattrfunc tp_getattr;
setattrfunc tp_setattr;
PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
or tp_reserved (Python 3) */

reprfunc tp_repr;

/* Method suites for standard classes */

PyNumberMethods *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods *tp_as_mapping;

/* More standard operations (here for binary compatibility) */

hashfunc tp_hash;
ternaryfunc tp_call;
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
unsigned long tp_flags;

const char *tp_doc; /* Documentation string */

/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
Py_ssize_t tp_weaklistoffset;

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;
destructor tp_finalize;

} PyTypeObject;
The type object structure extends the `PyVarObject` structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that `PyType_Type` (the metatype) initializes `tp_itemsize`, which means that its instances (i.e. type objects) must have the `ob_size` field.

**PyObject* `PyObject._ob_next`**
**PyObject* `PyObject._ob_prev`**

These fields are only present when the macro `Py_TRACE_REFS` is defined. Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain `NULL`. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of *all* live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

These fields are not inherited by subtypes.

**Py_ssize_t `PyObject.ob_refcnt`**

This is the type object's reference count, initialized to 1 by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

This field is not inherited by subtypes.

**PyTypeObject* `PyObject.ob_type`**

This is the type's type, in other words its metatype. It is initialized
by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```c
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. `PyType_Ready()` checks if `ob_type` is `NULL`, and if so, initializes it to the `ob_type` field of the base class. `PyType_Ready()` will not change this field if it is non-zero.

This field is inherited by subtypes.

`Py_ssize_t PyVarObject.ob_size`

For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field is not inherited by subtypes.

`const char* PyTypeObject.tp_name`

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in
module M in subpackage Q in package P should have the 
\texttt{tp\_name} initializer \texttt{"P.Q.M.T"}.

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key \texttt{"__module__"}.

For statically allocated type objects, the \texttt{tp\_name} field should contain a dot. Everything before the last dot is made accessible as the \texttt{__module__} attribute, and everything after the last dot is made accessible as the \texttt{__name__} attribute.

If no dot is present, the entire \texttt{tp\_name} field is made accessible as the \texttt{__name__} attribute, and the \texttt{__module__} attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field is not inherited by subtypes.

\texttt{Py\_ssize\_t \texttt{PyTypeObject\_tp\_basicsize}}

\texttt{Py\_ssize\_t \texttt{PyTypeObject\_tp\_itemsize}}

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero \texttt{tp\_itemsize} field, types with variable-length instances have a non-zero \texttt{tp\_itemsize} field. For a type with fixed-length instances, all instances have the same size, given in \texttt{tp\_basicsize}.

For a type with variable-length instances, the instances must have
an ob_size field, and the instance size is tp_basicsize plus N times tp_itemsize, where N is the “length” of the object. The value of N is typically stored in the instance's ob_size field. There are exceptions: for example, ints use a negative ob_size to indicate a negative number, and N is abs(ob_size) there. Also, the presence of an ob_size field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful ob_size field).

The basic size includes the fields in the instance declared by the macro PyObject_HEAD or PyObject_VAR_HEAD (whichever is used to declare the instance struct) and this in turn includes the _ob_prev and _ob_next fields if they are present. This means that the only correct way to get an initializer for the tp_basicsize is to use the sizeof operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

These fields are inherited separately by subtypes. If the base type has a non-zero tp_itemsize, it is generally not safe to set tp_itemsize to a different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of tp_basicsize. Example: suppose a type implements an array of double. tp_itemsize is sizeof(double). It is the programmer's responsibility that tp_basicsize is a multiple of sizeof(double) (assuming this is the alignment requirement for double).
**Destructor** `PyTypeObject.tp_dealloc`

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`).

The destructor function is called by the `Py_DECREF()` and `Py_XDECREF()` macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type’s `tp_free` function. If the type is not subtypable (doesn’t have the `Py_TPFLAGS_BASETYPE` flag bit set), it is permissible to call the object deallocator directly instead of via `tp_free`. The object deallocator should be the one used to allocate the instance; this is normally `PyObject_Del()` if the instance was allocated using `PyObject_New()` or `PyObject_VarNew()`, or `PyObject_GC_Del()` if the instance was allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.

This field is inherited by subtypes.

**Printfunc** `PyTypeObject.tp_print`

Reserved slot, formerly used for print formatting in Python 2.x.

**Getattrfunc** `PyTypeObject.tp_getattr`

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a
function that acts the same as the \texttt{tp\_getattro} function, but taking a C string instead of a Python string object to give the attribute name. The signature is

\begin{verbatim}
PyObject * tp_getattro(PyObject *o, char *attr_name)
\end{verbatim}

This field is inherited by subtypes together with \texttt{tp\_getattro}: a subtype inherits both \texttt{tp\_getattr} and \texttt{tp\_getattro} from its base type when the subtype's \texttt{tp\_getattr} and \texttt{tp\_getattro} are both \texttt{NULL}.

\texttt{setattrofuncPyTypeError Object.tp\_setattr}

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the \texttt{tp\_setattro} function, but taking a C string instead of a Python string object to give the attribute name. The signature is

\begin{verbatim}
PyObject * tp_setattr(PyObject *o, char *attr_name)
\end{verbatim}

The \texttt{v} argument is set to \texttt{NULL} to delete the attribute. This field is inherited by subtypes together with \texttt{tp\_setattro}: a subtype inherits both \texttt{tp\_setattr} and \texttt{tp\_setattro} from its base type when the subtype's \texttt{tp\_setattr} and \texttt{tp\_setattro} are both \texttt{NULL}.

\texttt{PyAsyncMethods\* tp\_as\_async}

Pointer to an additional structure that contains fields relevant only to objects which implement \texttt{awaitable} and \texttt{asynchronous iterator}
protocols at the C-level. See Async Object Structures for details.

*New in version 3.5:* Formerly known as `tp_compare` and `tp_reserved`.

**reprfunc** `PyTypeObject.tp_repr`

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for `PyObject_Repr()`; it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<` and ending with `'>` from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object’s memory address.

This field is inherited by subtypes.

**PyNumberMethods** `*tp_as_number`

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in Number Object Structures.

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

**PySequenceMethods** `*tp_as_sequence`

Pointer to an additional structure that contains fields relevant only
to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

**PyMappingMethods* `tp_as_mapping`**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

The `tp_as_mapping` field is not inherited, but the contained fields are inherited individually.

**`hashfunc` `PyTypeObject.tp_hash`**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for `PyObject_Hash();` it must return a value of the type `Py_hash_t`. The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

This field can be set explicitly to `PyObject_HashNotImplemented()` to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to `PyObject_HashNotImplemented()`.
When this field is not set, an attempt to take the hash of the object raises `TypeError`.

This field is inherited by subtypes together with `tp_richcompare`: a subtype inherits both of `tp_richcompare` and `tp_hash`, when the subtype’s `tp_richcompare` and `tp_hash` are both `NULL`.

ternaryfunc **PyTypeObject.tp_call**

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for `PyObject_Call()`. This field is inherited by subtypes.

reprfunc **PyTypeObject.tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls `PyObject_Str()` to do the actual work, and `PyObject_Str()` will call this handler.)

The signature is the same as for `PyObject_Str()`; it must return a string or a Unicode object. This function should return a “friendly” string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

When this field is not set, `PyObject_Repr()` is called to return a string representation. This field is inherited by subtypes.

getattrofunc **PyTypeObject.tp_getattro**

An optional pointer to the get-attribute function.
The signature is the same as for `PyObject_GetAttr()`. It is usually convenient to set this field to `PyObject_GenericGetAttr()`, which implements the normal way of looking for object attributes.

This field is inherited by subtypes together with `tp_getattr`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both `NULL`.

**setattrofunc** `PyTypeObject.tp_setattro`  
An optional pointer to the function for setting and deleting attributes.

The signature is the same as for `PyObject_SetAttr()`, but setting `v` to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to `PyObject_GenericSetAttr()`, which implements the normal way of setting object attributes.

This field is inherited by subtypes together with `tp_setattr`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

**PyBufferProcs* PyTypeObject.tp_as_buffer**  
Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.
unsigned long `PyTypeObject.tp_flags`

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or `NULL` value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the `tp_traverse` and `tp_clear` fields in the subtype exist and have `NULL` values.

The following bit masks are currently defined; these can be ORed together using the `|` operator to form the value of the `tp_flags` field. The macro `PyType_HasFeature()` takes a type and a flags value, `tp` and `f`, and checks whether `tp->tp_flags & f` is non-zero.

**Py_TPFLAGS_HEAPTYPE**

This bit is set when the type object itself is allocated on the heap. In this case, the `ob_type` field of its instances is considered a reference to the type, and the type object is
INCREF’ed when a new instance is created, and DECREF’ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance’s ob_type gets INCREF’ed or DECREF’ed).

**Py_TPFLAGS_BASETYPE**

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a “final” class in Java).

**Py_TPFLAGS_READY**

This bit is set when the type object has been fully initialized by `PyType_Ready()`.

**Py_TPFLAGS_READYING**

This bit is set while `PyType_Ready()` is in the process of initializing the type object.

**Py_TPFLAGS_HAVE_GC**

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using `PyObject_GC_New()` and destroyed using `PyObject_GC_Del()`. More information in section Supporting Cyclic Garbage Collection. This bit also implies that the GC-related fields `tp_traverse` and `tp_clear` are present in the type object.

**Py_TPFLAGS_DEFAULT**

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`,
Py_TPFLAGS_HAVE_VERSION_TAG

Py_TPFLAGS_LONG_SUBCLASS
Py_TPFLAGS_LIST_SUBCLASS
Py_TPFLAGS_TUPLE_SUBCLASS
Py_TPFLAGS_BYTES_SUBCLASS
Py_TPFLAGS_UNICODE_SUBCLASS
Py_TPFLAGS_DICT_SUBCLASS
Py_TPFLAGS_BASE_EXC_SUBCLASS
Py_TPFLAGS_TYPE_SUBCLASS

These flags are used by functions such as PyLong_Check() to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like PyObject_IsInstance(). Custom types that inherit from built-ins should have their tp_flags set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the tp_finalize slot is present in the type structure.

New in version 3.4.

const char* PyTypeObject.tp_doc

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the __doc__ attribute on the type and instances of the type.
This field is not inherited by subtypes.

**traverseproc** `PyTypeObject.tp_traverse`

An optional pointer to a traversal function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set. More information about Python’s garbage collection scheme can be found in section **Supporting Cyclic Garbage Collection**.

The `tp_traverse` pointer is used by the garbage collector to detect reference cycles. A typical implementation of a `tp_traverse` function simply calls `Py_VISIT()` on each of the instance's members that are Python objects. For example, this is function `local_traverse()` from the `_thread` extension module:

```c
static int local_traverse(localobject *self, visitproc visit
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that `Py_VISIT()` is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module’s `get_referents()` function will include it.
Note that `Py_VISIT()` requires the `visit` and `arg` parameters to local_traverse() to have these specific names; don’t name them just anything.

This field is inherited by subtypes together with `tp_clear` and the `Py_TPFLAGS_HAVE_GC` flag bit: the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

**Inquiry** `PyTypeObject.tp_clear`

An optional pointer to a clear function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set.

The `tp_clear` member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all `tp_clear` functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a `tp_clear` function. For example, the tuple type does not implement a `tp_clear` function, because it’s possible to prove that no reference cycle can be composed entirely of tuples. Therefore the `tp_clear` functions of other types must be sufficient to break any cycle containing a tuple. This isn’t immediately obvious, and there’s rarely a good reason to avoid implementing `tp_clear`.

Implementations of `tp_clear` should drop the instance’s references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```c
static int
local_clear(localobject *self)
```
The `Py_CLEAR()` macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to `NULL`. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference `self` again, it's important that the pointer to the contained object be `NULL` at that time, so that `self` knows the contained object can no longer be used. The `Py_CLEAR()` macro performs the operations in a safe order.

Because the goal of `tp_clear` functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's `tp_dealloc` function to invoke `tp_clear`.

More information about Python's garbage collection scheme can be found in section Supporting Cyclic Garbage Collection.

This field is inherited by subtypes together with `tp_traverse` and the `Py_TPFLAGS_HAVE_GC` flag bit: the flag bit, `tp_traverse`,
and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

### richcmpfunc `PyTypeObject.tp_richcompare`

An optional pointer to the rich comparison function, whose signature is `PyObject *tp_richcompare(PyObject *a, PyObject *b, int op)`. The first parameter is guaranteed to be an instance of the type that is defined by `PyTypeObject`.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

**Note:** If you want to implement a type for which only a limited set of comparisons makes sense (e.g. `==` and `!=`, but not `<` and friends), directly raise `TypeError` in the rich comparison function.

This field is inherited by subtypes together with `tp_hash`: a subtype inherits `tp_richcompare` and `tp_hash` when the subtype's `tp_richcompare` and `tp_hash` are both `NULL`.

The following constants are defined to be used as the third argument for `tp_richcompare` and for `PyObject_RichCompare()`:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Py_LT</td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>Py_LE</td>
<td><code>&lt;=</code></td>
</tr>
<tr>
<td>Py_EQ</td>
<td><code>==</code></td>
</tr>
</tbody>
</table>
The following macro is defined to ease writing rich comparison functions:

```c
PyObject *Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, int op)
```

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. `VAL_A` and `VAL_B` must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for `PyObject_RichCompare()`.

The return value’s reference count is properly incremented.

On error, sets an exception and returns NULL from the function.

*New in version 3.7.*

```c
Py_ssize_t PyTypeObject.tp_weaklistoffset
```

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*()` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to `NULL`.

Do not confuse this field with `tp_weaklist`; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses
a different weak reference list head than the base type. Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

`getiterfunc` **PyTypeObject.tp_iter**

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function).

This function has the same signature as `PyObject_GetIter()`.

This field is inherited by subtypes.

`iternextfunc` **PyTypeObject.tp_iternext**

An optional pointer to a function that returns the next item in an iterator. When the iterator is exhausted, it must return `NULL`; a `StopIteration` exception may or may not be set. When another
error occurs, it must return *NULL* too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

This field is inherited by subtypes.

struct *PyMethodDef* *PyTypeObject.tp_methods*

An optional pointer to a static *NULL*-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* *PyTypeObject.tp_members*

An optional pointer to a static *NULL*-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* *PyTypeObject.tp_getset*

An optional pointer to a static *NULL*-terminated array of
PyGetSetDef structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see tp_dict below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject* PyTypeObject.tp_base
An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to &PyBaseObject_Type (which to Python programmers is known as the type object).

PyObject* PyTypeObject.tp_dict
The type's dictionary is stored here by PyType_Ready().

This field should normally be initialized to NULL before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once PyType_Ready() has initialized the type, extra attributes for the type may be added to this dictionary only if they don’t correspond to overloaded operations (like __add__()).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Warning: It is not safe to use PyDict_SetItem() on or
otherwise modify `tp_dict` with the dictionary C-API.

descrgetfunc **PyTypeObject.tp_descr_get**
An optional pointer to a “descriptor get” function.

The function signature is

```c
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *dict)
```

This field is inherited by subtypes.

descrsetfunc **PyTypeObject.tp_descr_set**
An optional pointer to a function for setting and deleting a descriptor’s value.

The function signature is

```c
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *dict)
```

The `value` argument is set to `NULL` to delete the value. This field is inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_dictoffset**
If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict`; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset
from the start of the instance structure. If the value is less than zero, it specifies the offset from the end of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of str or tuple. Note that the tp_basicsize field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, tp_dictoffset should be set to -4 to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative tp_dictoffset as follows:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where tp_basicsize, tp_itemsize and tp_dictoffset are taken from the type object, and ob_size is taken from the instance. The absolute value is taken because ints use the sign of ob_size to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by _PyObject_GetDictPtr().)

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via tp_dictoffset, this should not be a problem.
When a type defined by a class statement has no __slots__ declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the tp_dictoffset is set to that slot's offset.

When a type defined by a class statement has a __slots__ declaration, the type inherits its tp_dictoffset from its base type.

(Adding a slot named __dict__ to the __slots__ declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like __weakref__ though.)

initproc PyTypeObject.tp_init
An optional pointer to an instance initialization function.

This function corresponds to the __init__() method of classes. Like __init__(), it is possible to create an instance without calling __init__() and it is possible to reinitialize an instance by calling its __init__() method again.

The function signature is

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds)
```

The self argument is the instance to be initialized; the args and kwds arguments represent positional and keyword arguments of the call to __init__().

The tp_init function, if not NULL, is called when an instance is created normally by calling its type, after the type's tp_new function has returned an instance of the type. If the tp_new
function returns an instance of some other type that is not a subtype of the original type, no tp_init function is called; if tp_new returns an instance of a subtype of the original type, the subtype's tp_init is called.

This field is inherited by subtypes.

allocfunc PyTypeObject.tp_alloc

An optional pointer to an instance allocation function.

The function signature is

```py
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t)
```

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with ob_refcnt set to 1 and ob_type set to the type argument. If the type's tp_itemsize is non-zero, the object's ob_size field should be initialized to nitems and the length of the allocated memory block should be tp_basicsize + nitems*tp_itemsize, rounded up to a multiple of sizeof(void*); otherwise, nitems is not used and the length of the block should be tp_basicsize.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by tp_new.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to PyType_GenericAlloc(), to force a
standard heap allocation strategy. That is also the recommended value for statically defined types.

**newfunc PyTypeObject.tp_new**

An optional pointer to an instance creation function.

If this function is *NULL* for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

The function signature is

```c
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```

The subtype argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn’t have to equal the type whose *tp_new* function is called; it may be a subtype of that type (but not an unrelated type).

The *tp_new* function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the *tp_init* handler. A good rule of thumb is that for immutable types, all initialization should take place in *tp_new*, while for mutable types, most initialization should be deferred to *tp_init*.

This field is inherited by subtypes, except it is not inherited by static types whose *tp_base* is *NULL* or &PyBaseObject_Type.

**destructor PyTypeObject.tp_free**
An optional pointer to an instance deallocation function. Its signature is `freefunc`:

```c
void tp_free(void *)
```

An initializer that is compatible with this signature is `PyObject_Free()`.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match `PyType_GenericAlloc()` and the value of the `Py_TPFLAGS_HAVE_GC` flag bit.

**inquiry** **`PyTypeObject.tp_is_gc`**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object’s type’s `tp_flags` field, and check the `Py_TPFLAGS_HAVE_GC` flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is

```c
int tp_is_gc(PyObject *self)
```

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes.
PyObject* **PyTypeObject.tp_bases**

Tuple of base types.

This is set for types created by a class statement. It should be *NULL* for statically defined types.

This field is not inherited.

PyObject* **PyTypeObject.tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with *object*, in Method Resolution Order.

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

destructor **PyTypeObject.tp_finalize**

An optional pointer to an instance finalization function. Its signature is *destructor*:

```c
void tp_finalize(PyObject *)
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

*tp_finalize* should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```c
static void
local_finalize(PyObject *self)
{
```
For this field to be taken into account (even through inheritance), you must also set the `Py_TPFLAGS_HAVE_FINALIZE` flags bit.

This field is inherited by subtypes.

*New in version 3.4.*

**See also:** “Safe object finalization” ([PEP 442](https://www.python.org/dev/peps/pep-0442/))

**PyObject** `PyTypeObject.tp_cache`

Unused. Not inherited. Internal use only.

**PyObject** `PyTypeObject.tp_subclasses`

List of weak references to subclasses. Not inherited. Internal use only.

**PyObject** `PyTypeObject.tp_weaklist`

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited...
by subtypes.

Py_ssize_t **PyTypeObject.tp_allocs**
Number of allocations.

Py_ssize_t **PyTypeObject.tp_frees**
Number of frees.

Py_ssize_t **PyTypeObject.tp_maxalloc**
Maximum simultaneously allocated objects.

**PyTypeObject** * PyTypeObject.tp_next*
Pointer to the next type object with a non-zero tp_allocs field.

Also, note that, in a garbage collected Python, tp_dealloc may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.
Number Object Structures

**PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the Number Protocol section.

Here is the structure definition:

```c
typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;
    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
} PyNumberMethods;
```
Note: Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return Py_NotImplemented, if another error occurred they must return NULL and set an exception.

Note: The nb_reserved field should always be NULL. It was previously called nb_long, and was renamed in Python 3.0.1.
Mapping Object Structures

PyMappingMethods
This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc PyMappingMethods.mp_length
This function is used by PyMapping_Size() and PyObject_Size(), and has the same signature. This slot may be set to NULL if the object has no defined length.

binaryfunc PyMappingMethods.mp_subscript
This function is used by PyObject_GetItem() and PySequence_GetSlice(), and has the same signature as PyObject_GetItem(). This slot must be filled for the PyMapping_Check() function to return 1, it can be NULL otherwise.

objobjargproc PyMappingMethods.mp_ass_subscript
This function is used by PyObject_SetItem(), PyObject_DelItem(), PyObject_SetSlice(), and PyObject_DelSlice(). It has the same signature as PyObject_SetItem(), but v can also be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.
Sequence Object Structures

PySequenceMethods
This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc PySequenceMethods.sq_length
This function is used by PySequence_Size() and PyObject_Size(), and has the same signature. It is also used for handling negative indices via the sq_item and the sq_ass_item slots.

binaryfunc PySequenceMethods.sq_concat
This function is used by PySequence_Concat() and has the same signature. It is also used by the + operator, after trying the numeric addition via the nb_add slot.

ssizeargfunc PySequenceMethods.sq_repeat
This function is used by PySequence_Repeat() and has the same signature. It is also used by the * operator, after trying numeric multiplication via the nb_multiply slot.

ssizeargfunc PySequenceMethods.sq_item
This function is used by PySequence_GetItem() and has the same signature. It is also used by PyObject_GetItem(), after trying the subscription via the mp_subscript slot. This slot must be filled for the PySequence_Check() function to return 1, it can be NULL otherwise.

Negative indexes are handled as follows: if the sq_length slot is
filled, it is called and the sequence length is used to compute a positive index which is passed to sq_item. If sq_length is NULL, the index is passed as is to the function.

**ssizeobjargproc PySequenceMethods.sq_ass_item**
This function is used by `PySequence_SetItem()` and has the same signature. It is also used by `PyObject_SetItem()` and `PyObject_DelItem()`, after trying the item assignment and deletion via the `mp_ass_subscript` slot. This slot may be left to NULL if the object does not support item assignment and deletion.

**objobjproc PySequenceMethods.sq_contains**
This function may be used by `PySequence_Contains()` and has the same signature. This slot may be left to NULL, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

**binaryfunc PySequenceMethods.sq_inplace_concat**
This function is used by `PySequence_InPlaceConcat()` and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case `PySequence_InPlaceConcat()` will fall back to `PySequence_Concat()`. It is also used by the augmented assignment `+=`, after trying numeric inplace addition via the `nb_inplace_add` slot.

**ssizeargfunc PySequenceMethods.sq_inplace_repeat**
This function is used by `PySequence_InPlaceRepeat()` and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case `PySequence_InPlaceRepeat()` will fall back to
PySequence_Repeat(). It is also used by the augmented assignment `*=` after trying numeric inplace multiplication via the nb_inplace_multiply slot.
Buffer Object Structures

PyBufferProcs

This structure holds pointers to the functions required by the Buffer protocol. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc PyBufferProcs.bf_getbuffer

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags)
```

Handle a request to exporter to fill in view as specified by flags. Except for point (3), an implementation of this function MUST take these steps:

1. Check if the request can be met. If not, raise PyExc_BufferError, set view->obj to NULL and return -1.
2. Fill in the requested fields.
3. Increment an internal counter for the number of exports.
4. Set view->obj to exporter and increment view->obj.
5. Return 0.

If exporter is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets view->obj to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, view->obj will be a new reference to the root object.
The individual fields of view are described in section Buffer structure, the rules how an exporter must react to specific requests are in section Buffer request types.

All memory pointed to in the Py_buffer structure belongs to the exporter and must remain valid until there are no consumers left. format, shape, strides, suboffsets and internal are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

releasebufferproc PyBufferProcs.bf_releasebuffer

The signature of this function is:

```c
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, PyBufferProcs.bf_releasebuffer may be NULL. Otherwise, a standard implementation of this function will take these optional steps:

1. Decrement an internal counter for the number of exports.
2. If the counter is 0, free all memory associated with view.

The exporter MUST use the internal field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the view argument.
This function MUST NOT decrement view->obj, since that is done automatically in `PyBuffer_Release()` (this scheme is useful for breaking reference cycles).

`PyBuffer_Release()` is the interface for the consumer that wraps this function.
Async Object Structures

New in version 3.5.

PyAsyncMethods
This structure holds pointers to the functions required to implement awaitable and asynchronous iterator objects.

Here is the structure definition:

```c
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

unaryfunc PyAsyncMethods.am_await
The signature of this function is:

```c
PyObject *am_await(PyObject *self)
```

The returned object must be an iterator, i.e. `PyIter_Check()` must return 1 for it.

This slot may be set to `NULL` if an object is not an awaitable.

unaryfunc PyAsyncMethods.am_aiter
The signature of this function is:

```c
PyObject *am_aiter(PyObject *self)
```

Must return an awaitable object. See `__anext__()` for details.
This slot may be set to \textit{NULL} if an object does not implement asynchronous iteration protocol.

unaryfunc \texttt{PyAsyncMethods.am\_anext}

The signature of this function is:

\begin{verbatim}
PyObject *am\_anext(PyObject *self)
\end{verbatim}

Must return an \textit{awaitable} object. See \texttt{\_\_anext\_\_()} for details. This slot may be set to \textit{NULL}.
Supporting Cyclic Garbage Collection

Python’s support for detecting and collecting garbage which involves circular references requires support from object types which are “containers” for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

**Py_TPFLAGS_HAVE_GC**

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

 Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

`TYPE* PyObject_GC_New(TYPE, PyTypeObject *type)`

Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.
TYPE* **PyObject_GC_NewVar**(TYPE, PyTypeObject *type, Py_ssize_t size)

Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **PyObject_GC_Resize**(TYPE, PyVarObject *op, Py_ssize_t newsize)

Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or `NULL` on failure. `op` must not be tracked by the collector yet.

void **PyObject_GC_Track**(PyObject *op)

Adds the object `op` to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

void **_PyObject_GC_TRACK**(PyObject *op)

A macro version of `PyObject_GC_Track()`. It should not be used for extension modules.

*Deprecated since version 3.6:* This macro is removed from Python 3.8.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.
2. The object’s memory must be deallocated using `PyObject_GC_Del()`.
void **PyObject**\_**GC\_Del**(void *\_op)
    Releases memory allocated to an object using **PyObject**\_**GC\_New**() or **PyObject**\_**GC\_NewVar**().

void **PyObject**\_**GC\_UnTrack**(void *\_op)
    Remove the object \_op from the set of container objects tracked by the collector. Note that **PyObject**\_**GC\_Track**() can be called again on this object to add it back to the set of tracked objects. The deallocator (**tp\_dealloc** handler) should call this for the object before any of the fields used by the **tp\_traverse** handler become invalid.

void **_PyObject**\_**GC\_UNTRACK**(PyObject *\_op)
    A macro version of **PyObject**\_**GC\_UnTrack**(). It should not be used for extension modules.

    *Deprecated since version 3.6:* This macro is removed from Python 3.8.

The **tp\_traverse** handler accepts a function parameter of this type:

int (**visitproc**)(PyObject *\_object, void *\_arg)
    Type of the visitor function passed to the **tp\_traverse** handler. The function should be called with an object to traverse as **object** and the third parameter to the **tp\_traverse** handler as **arg**. The Python core uses several visitor functions to implement cyclic garbage detection; it’s not expected that users will need to write their own visitor functions.

The **tp\_traverse** handler must have the following type:

int (**traverseproc**)(PyObject *\_self, visitproc **visit**, void *\_arg)
Traversing function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a *NULL* object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing *tp_traverse* handlers, a *Py_VISIT()* macro is provided. In order to use this macro, the *tp_traverse* implementation must name its arguments exactly *visit* and *arg*:

```c
void Py_VISIT(PyObject *o)
    If *o* is not *NULL*, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```c
static int my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The *tp_clear* handler must be of the *inquiry* type, or *NULL* if the object is immutable.

```c
int (*inquiry)(PyObject *self)
    Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don’t just call *Py_DECREF()* on a reference). The collector will call this method if
it detects that this object is involved in a reference cycle.
API and ABI Versioning

PY_VERSION_HEX is the Python version number encoded in a single integer.

For example if the PY_VERSION_HEX is set to \(0x030401a2\), the underlying version information can be found by treating it as a 32 bit number in the following manner:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits (big endian order)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-8</td>
<td>PY_MAJOR_VERSION (the 3 in 3.4.1a2)</td>
</tr>
<tr>
<td>2</td>
<td>9-16</td>
<td>PY_MINOR_VERSION (the 4 in 3.4.1a2)</td>
</tr>
<tr>
<td>3</td>
<td>17-24</td>
<td>PY_MICRO_VERSION (the 1 in 3.4.1a2)</td>
</tr>
<tr>
<td>4</td>
<td>25-28</td>
<td>PY_RELEASE_LEVEL (0xA for alpha, 0xB for beta, 0xC for release candidate and 0xF for final), in this case it is alpha.</td>
</tr>
<tr>
<td></td>
<td>29-32</td>
<td>PY_RELEASE_SERIAL (the 2 in 3.4.1a2, zero for final releases)</td>
</tr>
</tbody>
</table>

Thus 3.4.1a2 is hexversion \(0x030401a2\).

All the given macros are defined in Include/patchlevel.h.
Porting Python 2 Code to Python 3

author: Brett Cannon

Abstract

With Python 3 being the future of Python while Python 2 is still in active use, it is good to have your project available for both major releases of Python. This guide is meant to help you figure out how best to support both Python 2 & 3 simultaneously.

If you are looking to port an extension module instead of pure Python code, please see Porting Extension Modules to Python 3.

If you would like to read one core Python developer’s take on why Python 3 came into existence, you can read Nick Coghlan’s Python 3 Q & A or Brett Cannon’s Why Python 3 exists.

For help with porting, you can email the python-porting mailing list with questions.
The Short Explanation

To make your project be single-source Python 2/3 compatible, the basic steps are:

1. Only worry about supporting Python 2.7
2. Make sure you have good test coverage (coverage.py can help; pip install coverage)
3. Learn the differences between Python 2 & 3
4. Use Futurize (or Modernize) to update your code (e.g. pip install future)
5. Use Pylint to help make sure you don’t regress on your Python 3 support (pip install pylint)
6. Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (pip install caniusepython3)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; pip install tox)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3).
Details

A key point about supporting Python 2 & 3 simultaneously is that you can start **today**! Even if your dependencies are not supporting Python 3 yet that does not mean you can't modernize your code **now** to support Python 3. Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2 code.

Another key point is that modernizing your Python 2 code to also support Python 3 is largely automated for you. While you might have to make some API decisions thanks to Python 3 clarifying text data versus binary data, the lower-level work is now mostly done for you and thus can at least benefit from the automated changes immediately.

Keep those key points in mind while you read on about the details of porting your code to support Python 2 & 3 simultaneously.

**Drop support for Python 2.6 and older**

While you can make Python 2.5 work with Python 3, it is **much** easier if you only have to work with Python 2.7. If dropping Python 2.5 is not an option then the **six** project can help you support Python 2.5 & 3 simultaneously (**pip install six**). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

If you are able to skip Python 2.5 and older, then the required changes to your code should continue to look and feel like idiomatic Python code. At worst you will have to use a function instead of a method in some instances or have to import a function instead of using a built-in one, but otherwise the overall transformation should not feel foreign to you.
But you should aim for only supporting Python 2.7. Python 2.6 is no longer freely supported and thus is not receiving bugfixes. This means **you** will have to work around any issues you come across with Python 2.6. There are also some tools mentioned in this HOWTO which do not support Python 2.6 (e.g., **Pylint**), and this will become more commonplace as time goes on. It will simply be easier for you if you only support the versions of Python that you have to support.

**Make sure you specify the proper version support in your setup.py file**

In your **setup.py** file you should have the proper **trove classifier** specifying what versions of Python you support. As your project does not support Python 3 yet you should at least have **Programming Language :: Python :: 2 :: Only** specified. Ideally you should also specify each major/minor version of Python that you do support, e.g. **Programming Language :: Python :: 2.7**.

**Have good test coverage**

Once you have your code supporting the oldest version of Python 2 you want it to, you will want to make sure your test suite has good coverage. A good rule of thumb is that if you want to be confident enough in your test suite that any failures that appear after having tools rewrite your code are actual bugs in the tools and not in your code. If you want a number to aim for, try to get over 80% coverage (and don't feel bad if you find it hard to get better than 90% coverage). If you don't already have a tool to measure test coverage then **coverage.py** is recommended.
Learn the differences between Python 2 & 3

Once you have your code well-tested you are ready to begin porting your code to Python 3! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2. Typically the two best ways of doing that is reading the “What’s New” doc for each release of Python 3 and the Porting to Python 3 book (which is free online). There is also a handy cheat sheet from the Python-Future project.

Update your code

Once you feel like you know what is different in Python 3 compared to Python 2, it’s time to update your code! You have a choice between two tools in porting your code automatically: Futurize and Modernize. Which tool you choose will depend on how much like Python 3 you want your code to be. Futurize does its best to make Python 3 idioms and practices exist in Python 2, e.g. backporting the bytes type from Python 3 so that you have semantic parity between the major versions of Python. Modernize, on the other hand, is more conservative and targets a Python 2/3 subset of Python, directly relying on six to help provide compatibility. As Python 3 is the future, it might be best to consider Futurize to begin adjusting to any new practices that Python 3 introduces which you are not accustomed to yet.

Regardless of which tool you choose, they will update your code to run under Python 3 while staying compatible with the version of Python 2 you started with. Depending on how conservative you want to be, you may want to run the tool over your test suite first and visually inspect the diff to make sure the transformation is accurate. After you have
transformed your test suite and verified that all the tests still pass as expected, then you can transform your application code knowing that any tests which fail is a translation failure.

Unfortunately the tools can't automate everything to make your code work under Python 3 and so there are a handful of things you will need to update manually to get full Python 3 support (which of these steps are necessary vary between the tools). Read the documentation for the tool you choose to use to see what it fixes by default and what it can do optionally to know what will (not) be fixed for you and what you may have to fix on your own (e.g. using `io.open()` over the built-in `open()` function is off by default in Modernize). Luckily, though, there are only a couple of things to watch out for which can be considered large issues that may be hard to debug if not watched for.

**Division**

In Python 3, `5 / 2 == 2.5` and not `2`; all division between `int` values result in a `float`. This change has actually been planned since Python 2.2 which was released in 2002. Since then users have been encouraged to add `from __future__ import division` to any and all files which use the `/` and `//` operators or to be running the interpreter with the `-Q` flag. If you have not been doing this then you will need to go through your code and do two things:

1. Add `from __future__ import division` to your files
2. Update any division operator as necessary to either use `//` to use floor division or continue using `/` and expect a float

The reason that `/` isn't simply translated to `//` automatically is that if an object defines a `__truediv__` method but not `__floordiv__` then your code would begin to fail (e.g. a user-defined class that uses
Text versus binary data

In Python 2 you could use the `str` type for both text and binary data. Unfortunately this confluence of two different concepts could lead to brittle code which sometimes worked for either kind of data, sometimes not. It also could lead to confusing APIs if people didn't explicitly state that something that accepted `str` accepted either text or binary data instead of one specific type. This complicated the situation especially for anyone supporting multiple languages as APIs wouldn't bother explicitly supporting `unicode` when they claimed text data support.

To make the distinction between text and binary data clearer and more pronounced, Python 3 did what most languages created in the age of the internet have done and made text and binary data distinct types that cannot blindly be mixed together (Python predates widespread access to the internet). For any code that deals only with text or only binary data, this separation doesn't pose an issue. But for code that has to deal with both, it does mean you might have to now care about when you are using text compared to binary data, which is why this cannot be entirely automated.

To start, you will need to decide which APIs take text and which take binary (it is **highly** recommended you don't design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` and those that work with binary data work with the `bytes` type from Python 3 (which is a subset of `str` in Python 2 and acts as an alias for `bytes` type in Python 2). Usually the biggest issue is realizing which methods exist on which
types in Python 2 & 3 simultaneously (for text that's `unicode` in Python 2 and `str` in Python 3, for binary that's `str/bytes` in Python 2 and `bytes` in Python 3). The following table lists the unique methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can’t be used by the textual data type consistently between Python 2 and 3 because `str` in Python 3 doesn't have the method). Do note that as of Python 3.5 the `__mod__` method was added to the bytes type.

<table>
<thead>
<tr>
<th>Text data</th>
<th>Binary data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decode</td>
</tr>
<tr>
<td>encode</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td></td>
</tr>
<tr>
<td>isdecimal</td>
<td></td>
</tr>
<tr>
<td>isnumeric</td>
<td></td>
</tr>
</tbody>
</table>

Making the distinction easier to handle can be accomplished by encoding and decoding between binary data and text at the edge of your code. This means that when you receive text in binary data, you should immediately decode it. And if your code needs to send text as binary data then encode it as late as possible. This allows your code to work with only text internally and thus eliminates having to keep track of what type of data you are working with.

The next issue is making sure you know whether the string literals in your code represent text or binary data. You should add a `b` prefix to any literal that presents binary data. For text you should add a `u` prefix to the text literal. (there is a `__future__` import to force all
unspecified literals to be Unicode, but usage has shown it isn’t as effective as adding a b or u prefix to all literals explicitly)

As part of this dichotomy you also need to be careful about opening files. Unless you have been working on Windows, there is a chance you have not always bothered to add the b mode when opening a binary file (e.g., rb for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the io module for details. Therefore, you **must** make a decision of whether a file will be used for binary access (allowing binary data to be read and/or written) or textual access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it's actually `io.open()`). Do not bother with the outdated practice of using `codecs.open()` as that's only necessary for keeping compatibility with Python 2.5.

The constructors of both `str` and `bytes` have different semantics for the same arguments between Python 2 & 3. Passing an integer to `bytes` in Python 2 will give you the string representation of the integer: `bytes(3) == '3'`. But in Python 3, an integer argument to `bytes` will give you a bytes object as long as the integer specified, filled with null bytes: `bytes(3) == b'\x00\x00\x00'`. A similar worry is necessary when passing a bytes object to `str`. In Python 2 you just get the bytes object back: `str(b'3') == b'3'`. But in Python 3 you get the string representation of the bytes object: `str(b'3') == "b'3'"`.

Finally, the indexing of binary data requires careful handling (slicing does **not** require any special handling). In Python 2, `b'123'[1] ==`
b'2' while in Python 3 b'123'[1] == 50. Because binary data is simply a collection of binary numbers, Python 3 returns the integer value for the byte you index on. But in Python 2 because bytes == str, indexing returns a one-item slice of bytes. The six project has a function named six.indexbytes() which will return an integer like in Python 3: six.indexbytes(b'123', 1).

To summarize:

1. Decide which of your APIs take text and which take binary data
2. Make sure that your code that works with text also works with unicode and code for binary data works with bytes in Python 2 (see the table above for what methods you cannot use for each type)
3. Mark all binary literals with a b prefix, textual literals with a u prefix
4. Decode binary data to text as soon as possible, encode text as binary data as late as possible
5. Open files using io.open() and make sure to specify the b mode when appropriate
6. Be careful when indexing into binary data

Use feature detection instead of version detection

Inevitably you will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python you’re running under supports what you need. If for some reason that doesn’t work then you should make the version check be against Python 2 and not Python 3. To help explain this, let’s look at an example.

Let’s pretend that you need access to a feature of importlib that is available in Python’s standard library since Python 3.3 and available
for Python 2 through `importlib2` on PyPI. You might be tempted to write code to access e.g. the `importlib.abc` module by doing the following:

```python
import sys
if sys.version_info[0] == 3:
    from importlib import abc
else:
    from importlib2 import abc
```

The problem with this code is what happens when Python 4 comes out? It would be better to treat Python 2 as the exceptional case instead of Python 3 and assume that future Python versions will be more compatible with Python 3 than Python 2:

```python
import sys
if sys.version_info[0] > 2:
    from importlib import abc
else:
    from importlib2 import abc
```

The best solution, though, is to do no version detection at all and instead rely on feature detection. That avoids any potential issues of getting the version detection wrong and helps keep you future-compatible:

```python
try:
    from importlib import abc
except ImportError:
    from importlib2 import abc
```

Prevent compatibility regressions
Once you have fully translated your code to be compatible with Python 3, you will want to make sure your code doesn’t regress and stop working under Python 3. This is especially true if you have a dependency which is blocking you from actually running under Python 3 at the moment.

To help with staying compatible, any new modules you create should have at least the following block of code at the top of it:

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

You can also run Python 2 with the `-3` flag to be warned about various compatibility issues your code triggers during execution. If you turn warnings into errors with `-Werror` then you can make sure that you don’t accidentally miss a warning.

You can also use the Pylint project and its `--py3k` flag to lint your code to receive warnings when your code begins to deviate from Python 3 compatibility. This also prevents you from having to run Modernize or Futurize over your code regularly to catch compatibility regressions. This does require you only support Python 2.7 and Python 3.4 or newer as that is Pylint’s minimum Python version support.

**Check which dependencies block your transition**

**After** you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also been ported. The caniusepython3 project was created to help you determine which projects – directly or indirectly – are blocking you from
supporting Python 3. There is both a command-line tool as well as a web interface at https://caniusepython3.com.

The project also provides code which you can integrate into your test suite so that you will have a failing test when you no longer have dependencies blocking you from using Python 3. This allows you to avoid having to manually check your dependencies and to be notified quickly when you can start running on Python 3.

**Update your setup.py file to denote Python 3 compatibility**

Once your code works under Python 3, you should update the classifiers in your setup.py to contain *Programming Language :: Python :: 3* and to not specify sole Python 2 support. This will tell anyone using your code that you support Python 2 and 3. Ideally you will also want to add classifiers for each major/minor version of Python you now support.

**Use continuous integration to stay compatible**

Once you are able to fully run under Python 3 you will want to make sure your code always works under both Python 2 & 3. Probably the best tool for running your tests under multiple Python interpreters is *tox*. You can then integrate tox with your continuous integration system so that you never accidentally break Python 2 or 3 support.

You may also want to use the `-bb` flag with the Python 3 interpreter to trigger an exception when you are comparing bytes to strings or bytes to an int (the latter is available starting in Python 3.5). By default type-differing comparisons simply return `False`, but if you made a mistake
in your separation of text/binary data handling or indexing on bytes you wouldn’t easily find the mistake. This flag will raise an exception when these kinds of comparisons occur, making the mistake much easier to track down.

And that’s mostly it! At this point your code base is compatible with both Python 2 and 3 simultaneously. Your testing will also be set up so that you don’t accidentally break Python 2 or 3 compatibility regardless of which version you typically run your tests under while developing.

Consider using optional static type checking

Another way to help port your code is to use a static type checker like mypy or pytype on your code. These tools can be used to analyze your code as if it’s being run under Python 2, then you can run the tool a second time as if your code is running under Python 3. By running a static type checker twice like this you can discover if you’re e.g. misusing binary data type in one version of Python compared to another. If you add optional type hints to your code you can also explicitly state whether your APIs use textual or binary data, helping to make sure everything functions as expected in both versions of Python.
Porting Extension Modules to Python 3

author: Benjamin Peterson

Abstract

Although changing the C-API was not one of Python 3’s objectives, the many Python-level changes made leaving Python 2’s API intact impossible. In fact, some changes such as `int()` and `long()` unification are more obvious on the C level. This document endeavors to document incompatibilities and how they can be worked around.
Conditional compilation

The easiest way to compile only some code for Python 3 is to check if PY_MAJOR_VERSION is greater than or equal to 3.

```c
#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif
```

API functions that are not present can be aliased to their equivalents within conditional blocks.
Changes to Object APIs

Python 3 merged together some types with similar functions while cleanly separating others.

str/unicode Unification

Python 3's `str()` type is equivalent to Python 2's `unicode()`; the C functions are called `PyUnicode_*` for both. The old 8-bit string type has become `bytes()`, with C functions called `PyBytes_*`. Python 2.6 and later provide a compatibility header, `bytesobject.h`, mapping `PyBytes` names to `PyString` ones. For best compatibility with Python 3, `PyUnicode` should be used for textual data and `PyBytes` for binary data. It's also important to remember that `PyBytes` and `PyUnicode` in Python 3 are not interchangeable like `PyString` and `PyUnicode` are in Python 2. The following example shows best practices with regards to `PyUnicode`, `PyString`, and `PyBytes`.

```c
#include "stdlib.h"
#include "Python.h"
#include "bytesobject.h"

/* text example */
static PyObject *
say_hello(PyObject *self, PyObject *args) {
    PyObject *name, *result;

    if (!PyArg_ParseTuple(args, "U:say_hello", &name)
        return NULL;

    result = PyUnicode_FromFormat("Hello, %S!", name)
```
return result;
}

/* just a forward */
static char * do_encode(PyObject *);

/* bytes example */
static PyObject * encode_object(PyObject *self, PyObject *args) {
    char *encoded;
    PyObject *result, *myobj;

    if (!PyArg_ParseTuple(args, "O:encode_object", &
        return NULL;

    encoded = do_encode(myobj);
    if (encoded == NULL)
        return NULL;
    result = PyBytes_FromString(encoded);
    free(encoded);
    return result;
}

long/int Unification

Python 3 has only one integer type, int(). But it actually corresponds to Python 2's long() type—the int() type used in Python 2 was removed. In the C-API, PyInt_* functions are replaced by their PyLong_* equivalents.
Module initialization and state

Python 3 has a revamped extension module initialization system. (See PEP 3121.) Instead of storing module state in globals, they should be stored in an interpreter specific structure. Creating modules that act correctly in both Python 2 and Python 3 is tricky. The following simple example demonstrates how.

```c
#include "Python.h"

struct module_state {
    PyObject *error;
};

#if PY_MAJOR_VERSION >= 3
#define GETSTATE(m) ((struct module_state*)PyModule_GetState(m))
#else
#define GETSTATE(m) (&_state)
#endif

static struct module_state _state;

static PyObject *
error_out(PyObject *m) {
    struct module_state *st = GETSTATE(m);
    PyErr_SetString(st->error, "something bad happened"
    return NULL;
}

static PyMethodDef myextension_methods[] = {
    {"error_out", (PyCFunction)error_out, METH_NOARGS}
};

#if PY_MAJOR_VERSION >= 3
```
static int myextension_traverse(PyObject *m, visitproc visitp) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}

static int myextension_clear(PyObject *m) {
    Py_CLEAR(GETSTATE(m)->error);
    return 0;
}

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "myextension",
    NULL,
    sizeof(struct module_state),
    myextension_methods,
    NULL,
    myextension_traverse,
    myextension_clear,
    NULL
};

#define INITERROR return NULL

PyMODINIT_FUNC
PyInit_myextension(void)
{
    #else
    #define INITERROR return
    void
    initmyextension(void)
    #endif
    {
    #if PY_MAJOR_VERSION >= 3
        PyObject *module = PyModule_Create(&moduledef);
        return module;
    }
    else
    {
        moduledef = Py_InitModule4("myextension", myextension_methods, NULL, NULL);
        return module;
    }

    return INITERROR;
}
#else

PyObject *module = Py_InitModule("myextension",
#endif

if (module == NULL)
    INITERROR;
struct module_state *st = GETSTATE(module);

st->error = PyErr_NewException("myextension.Error",
    if (st->error == NULL) {
        Py_DECREF(module);
        INITERROR;
    }
#endif

    if PY_MAJOR_VERSION >= 3
    return module;
#endif
CObject replaced with Capsule

The Capsule object was introduced in Python 3.1 and 2.7 to replace CObject. CObjects were useful, but the CObject API was problematic: it didn’t permit distinguishing between valid CObjects, which allowed mismatched CObjects to crash the interpreter, and some of its APIs relied on undefined behavior in C. (For further reading on the rationale behind Capsules, please see bpo-5630.)

If you’re currently using CObjects, and you want to migrate to 3.1 or newer, you’ll need to switch to Capsules. CObject was deprecated in 3.1 and 2.7 and completely removed in Python 3.2. If you only support 2.7, or 3.1 and above, you can simply switch to Capsule. If you need to support Python 3.0, or versions of Python earlier than 2.7, you’ll have to support both CObjects and Capsules. (Note that Python 3.0 is no longer supported, and it is not recommended for production use.)

The following example header file capsulethunk.h may solve the problem for you. Simply write your code against the Capsule API and include this header file after Python.h. Your code will automatically use Capsules in versions of Python with Capsules, and switch to CObjects when Capsules are unavailable.

capsulethunk.h simulates Capsules using CObjects. However, CObject provides no place to store the capsule’s “name”. As a result the simulated Capsule objects created by capsulethunk.h behave slightly differently from real Capsules. Specifically:

- The name parameter passed in to PyCapsule_New() is ignored.
- The name parameter passed in to PyCapsule_IsValid()
and `PyCapsule_GetPointer()` is ignored, and no error checking of the name is performed.

- `PyCapsule_GetName()` always returns NULL.
- `PyCapsule_SetName()` always raises an exception and returns failure. (Since there’s no way to store a name in a CObject, noisy failure of `PyCapsule_SetName()` was deemed preferable to silent failure here. If this is inconvenient, feel free to modify your local copy as you see fit.)

You can find `capsulethunk.h` in the Python source distribution as `Doc/includes/capsulethunk.h`. We also include it here for your convenience:

```c
#ifndef __CAPSULETHUNK_H
#define __CAPSULETHUNK_H

#if (   (PY_VERSION_HEX < 0x02070000) \n    || ((PY_VERSION_HEX >= 0x03000000) \n        && (PY_VERSION_HEX < 0x03010000)))

#define __PyCapsule_GetField(capsule, field, default_value) 
    (((PyCObject *)capsule)->field)
#define __PyCapsule_SetField(capsule, field, value)
    (((PyCObject *)capsule)->field = value), 1

#define PyCapsule_Type PyCObject_Type
```

/* Don't call PyCObject_SetPointer here, it fails if */
#define PyCapsule_SetPointer(capsule, pointer)  
    __PyCapsule_SetField(capsule, cobject, pointer)

#define PyCapsule_GetDestructor(capsule)  
    __PyCapsule_GetField(capsule, destructor)

#define PyCapsule_SetDestructor(capsule, dtor)  
    __PyCapsule_SetField(capsule, destructor, dtor)

/*
 * Sorry, there's simply no place
 * to store a Capsule "name" in a CObject.
 */
#define PyCapsule.GetName(capsule) NULL

static int
PyCapsule_SetName(PyObject *capsule, const char *unused
{
    unused = unused;
    PyErr_SetString(PyExc_NotImplementedError,
        "can't use PyCapsule_SetName with CObjects")
    return 1;
}
#define PyCapsule_GetContext(capsule) \ 
    __PyCapsule_GetField(capsule, descr)

#define PyCapsule_SetContext(capsule, context) \ 
    __PyCapsule_SetField(capsule, descr, context)

static void * 
PyCapsule_Import(const char *name, int no_block) 
{
    PyObject *object = NULL;
    void *return_value = NULL;
    char *trace;
    size_t name_length = (strlen(name) + 1) * sizeof(char);
    char *name_dup = (char *)PyMem_MALLOC(name_length);

    if (!name_dup) {
        return NULL;
    }

    memcpy(name_dup, name, name_length);

    trace = name_dup;
    while (trace) {
        char *dot = strchr(trace, '.');
        if (dot) {
            *dot++ = '\0';
        }
    }

    if (object == NULL) {
        if (no_block) {
            object = PyImport_ImportModuleNoBlock
        } else {
            
        } 
    
    
}
object = PyImport_ImportModule(trace)

if (!object) {
    PyErr_Format(PyExc_ImportError,
                 "PyCapsule_Import could not import module \"%s\"", trace)
}

else {
    PyObject *object2 = PyObject_GetAttrString
    Py_DECREF(object);
    object = object2;
}

if (!object) {
    goto EXIT;
}

trace = dot;
}

if (PyCObject_Check(object)) {
    PyCObject *cobject = (PyCObject *)object;
    return_value = cobject->cobject;
} else {
    PyErr_Format(PyExc_AttributeError,
                 "PyCapsule_Import \"%s\" is not valid", name);
}

EXIT:

Py_XDECREF(object);
if (name_dup) {
    PyMem_FREE(name_dup);
}
return return_value;

#endif /* #if PY_VERSION_HEX < 0x02070000 */
#endif /* __CAPSULETHUNK_H */
Other options

If you are writing a new extension module, you might consider Cython. It translates a Python-like language to C. The extension modules it creates are compatible with Python 3 and Python 2.
Curses Programming with Python

<table>
<thead>
<tr>
<th>Author:</th>
<th>A.M. Kuchling, Eric S. Raymond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release:</td>
<td>2.04</td>
</tr>
</tbody>
</table>

**Abstract**

This document describes how to use the `curses` extension module to control text-mode displays.
What is curses?

The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminal provided by various programs. Display terminals support various control codes to perform common operations such as moving the cursor, scrolling the screen, and erasing areas. Different terminals use widely differing codes, and often have their own minor quirks.

In a world of graphical displays, one might ask “why bother”? It’s true that character-cell display terminals are an obsolete technology, but there are niches in which being able to do fancy things with them are still valuable. One niche is on small-footprint or embedded Unixes that don’t run an X server. Another is tools such as OS installers and kernel configurators that may have to run before any graphical support is available.

The curses library provides fairly basic functionality, providing the programmer with an abstraction of a display containing multiple non-overlapping windows of text. The contents of a window can be changed in various ways—adding text, erasing it, changing its appearance—and the curses library will figure out what control codes need to be sent to the terminal to produce the right output. curses doesn’t provide many user-interface concepts such as buttons, checkboxes, or dialogs; if you need such features, consider a user interface library such as Urwid.

The curses library was originally written for BSD Unix; the later System V versions of Unix from AT&T added many enhancements and new functions. BSD curses is no longer maintained, having been replaced by ncurses, which is an open-source implementation of the AT&T
interface. If you’re using an open-source Unix such as Linux or FreeBSD, your system almost certainly uses ncurses. Since most current commercial Unix versions are based on System V code, all the functions described here will probably be available. The older versions of curses carried by some proprietary Unixes may not support everything, though.

The Windows version of Python doesn’t include the `curses` module. A ported version called `UniCurses` is available. You could also try the `Console module` written by Fredrik Lundh, which doesn’t use the same API as curses but provides cursor-addressable text output and full support for mouse and keyboard input.

**The Python curses module**

The Python module is a fairly simple wrapper over the C functions provided by curses; if you’re already familiar with curses programming in C, it’s really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler by merging different C functions such as `addstr()`, `mvaddstr()`, and `mvwaddstr()` into a single `addstr()` method. You’ll see this covered in more detail later.

This HOWTO is an introduction to writing text-mode programs with curses and Python. It doesn’t attempt to be a complete guide to the curses API; for that, see the Python library guide’s section on ncurses, and the C manual pages for ncurses. It will, however, give you the basic ideas.
Starting and ending a curses application

Before doing anything, curses must be initialized. This is done by calling the `initscr()` function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, `initscr()` returns a window object representing the entire screen; this is usually called `stdscr` after the name of the corresponding C variable.

```python
import curses
stdscr = curses.initscr()
```

Usually curses applications turn off automatic echoing of keys to the screen, in order to be able to read keys and only display them under certain circumstances. This requires calling the `noecho()` function.

```python
curses.noecho()
```

Applications will also commonly need to react to keys instantly, without requiring the Enter key to be pressed; this is called cbreak mode, as opposed to the usual buffered input mode.

```python
curses.cbreak()
```

Terminals usually return special keys, such as the cursor keys or navigation keys such as Page Up and Home, as a multibyte escape sequence. While you could write your application to expect such sequences and process them accordingly, curses can do it for you, returning a special value such as `curses.KEY_LEFT`. To get curses to do the job, you’ll have to enable keypad mode.

```python
stdscr.keypad(True)
```
Terminating a curses application is much easier than starting one. You'll need to call:

```python
curses.nocbreak()
stdscr.keypad(False)
curses.echo()
```
to reverse the curses-friendly terminal settings. Then call the `endwin()` function to restore the terminal to its original operating mode.

```python
curses.endwin()
```

A common problem when debugging a curses application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

In Python you can avoid these complications and make debugging much easier by importing the `curses.wrapper()` function and using it like this:

```python
from curses import wrapper

def main(stdscr):
    # Clear screen
    stdscr.clear()

    # This raises ZeroDivisionError when i == 10.
    for i in range(0, 11):
        v = i-10
```
The `wrapper()` function takes a callable object and does the initializations described above, also initializing colors if color support is present. `wrapper()` then runs your provided callable. Once the callable returns, `wrapper()` will restore the original state of the terminal. The callable is called inside a `try...except` that catches exceptions, restores the state of the terminal, and then re-raises the exception. Therefore your terminal won't be left in a funny state on exception and you'll be able to read the exception's message and traceback.
Windows and Pads

Windows are the basic abstraction in curses. A window object represents a rectangular area of the screen, and supports methods to display text, erase it, allow the user to input strings, and so forth.

The `stdscr` object returned by the `initscr()` function is a window object that covers the entire screen. Many programs may need only this single window, but you might wish to divide the screen into smaller windows, in order to redraw or clear them separately. The `newwin()` function creates a new window of a given size, returning the new window object.

```py
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

Note that the coordinate system used in curses is unusual. Coordinates are always passed in the order y,x, and the top-left corner of a window is coordinate (0,0). This breaks the normal convention for handling coordinates where the x coordinate comes first. This is an unfortunate difference from most other computer applications, but it's been part of curses since it was first written, and it's too late to change things now.

Your application can determine the size of the screen by using the `curses.LINES` and `curses.COLS` variables to obtain the y and x sizes. Legal coordinates will then extend from `(0,0)` to `(curses.LINES - 1, curses.COLS - 1)`.

When you call a method to display or erase text, the effect doesn't
immediately show up on the display. Instead you must call the `refresh()` method of window objects to update the screen.

This is because curses was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen was very important. Instead curses accumulates changes to the screen and displays them in the most efficient manner when you call `refresh()`. For example, if your program displays some text in a window and then clears the window, there’s no need to send the original text because they’re never visible.

In practice, explicitly telling curses to redraw a window doesn’t really complicate programming with curses much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by first calling `stdscr.refresh()` or the `refresh()` method of some other relevant window.

A pad is a special case of a window; it can be larger than the actual display screen, and only a portion of the pad displayed at a time. Creating a pad requires the pad’s height and width, while refreshing a pad requires giving the coordinates of the on-screen area where a subsection of the pad will be displayed.

```python
pad = curses.newpad(100, 100)
# These loops fill the pad with letters; addch() is # explained in the next section
for y in range(0, 99):
    for x in range(0, 99):
        pad.addch(y, x, ord('a') + (x*x+y*y) % 26)

# Displays a section of the pad in the middle of the # (0,0) : coordinate of upper-left corner of pad are
```
The `refresh()` call displays a section of the pad in the rectangle extending from coordinate (5,5) to coordinate (20,75) on the screen; the upper left corner of the displayed section is coordinate (0,0) on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to update the screen and prevent annoying screen flicker as each part of the screen gets updated. `refresh()` actually does two things:

1. Calls the `noutrefresh()` method of each window to update an underlying data structure representing the desired state of the screen.
2. Calls the function `doupdate()` function to change the physical screen to match the desired state recorded in the data structure.

Instead you can call `noutrefresh()` on a number of windows to update the data structure, and then call `doupdate()` to update the screen.
Displaying Text

From a C programmer’s point of view, curses may sometimes look like a twisty maze of functions, all subtly different. For example, `addstr()` displays a string at the current cursor location in the `stdscr` window, while `mvaddstr()` moves to a given `y,x` coordinate first before displaying the string. `waddstr()` is just like `addstr()`, but allows specifying a window to use instead of using `stdscr` by default. `mvwaddstr()` allows specifying both a window and a coordinate.

Fortunately the Python interface hides all these details. `stdscr` is a window object like any other, and methods such as `addstr()` accept multiple argument forms. Usually there are four different forms.

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str</code> or <code>ch</code></td>
<td>Display the string <code>str</code> or character <code>ch</code> at the current position</td>
</tr>
<tr>
<td><code>str</code> or <code>ch</code>, <code>attr</code></td>
<td>Display the string <code>str</code> or character <code>ch</code>, using attribute <code>attr</code> at the current position</td>
</tr>
<tr>
<td><code>y</code>, <code>x</code>, <code>str</code> or <code>ch</code></td>
<td>Move to position <code>y</code>, <code>x</code> within the window, and display <code>str</code> or <code>ch</code></td>
</tr>
<tr>
<td><code>y</code>, <code>x</code>, <code>str</code> or <code>ch</code>, <code>attr</code></td>
<td>Move to position <code>y</code>, <code>x</code> within the window, and display <code>str</code> or <code>ch</code>, using attribute <code>attr</code></td>
</tr>
</tbody>
</table>

Attributes allow displaying text in highlighted forms such as boldface, underline, reverse code, or in color. They’ll be explained in more detail in the next subsection.
The `addstr()` method takes a Python string or bytestring as the value to be displayed. The contents of bytestrings are sent to the terminal as-is. Strings are encoded to bytes using the value of the window's encoding attribute; this defaults to the default system encoding as returned by `locale.getpreferredencoding()`.

The `addch()` methods take a character, which can be either a string of length 1, a bytestring of length 1, or an integer.

Constants are provided for extension characters; these constants are integers greater than 255. For example, ACS_PLMINUS is a +/- symbol, and ACS_ULCORNER is the upper left corner of a box (handy for drawing borders). You can also use the appropriate Unicode character.

Windows remember where the cursor was left after the last operation, so if you leave out the `y,x` coordinates, the string or character will be displayed wherever the last operation left off. You can also move the cursor with the `move(y,x)` method. Because some terminals always display a flashing cursor, you may want to ensure that the cursor is positioned in some location where it won’t be distracting; it can be confusing to have the cursor blinking at some apparently random location.

If your application doesn’t need a blinking cursor at all, you can call `curs_set(False)` to make it invisible. For compatibility with older curses versions, there’s a `leaveok(bool)` function that’s a synonym for `curs_set()`. When `bool` is true, the curses library will attempt to suppress the flashing cursor, and you won’t need to worry about leaving it in odd locations.

Attributes and Color
Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video, or a text viewer may need to highlight certain words. curses supports this by allowing you to specify an attribute for each cell on the screen.

An attribute is an integer, each bit representing a different attribute. You can try to display text with multiple attribute bits set, but curses doesn’t guarantee that all the possible combinations are available, or that they’re all visually distinct. That depends on the ability of the terminal being used, so it’s safest to stick to the most commonly available attributes, listed here.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_BLINK</td>
<td>Blinking text</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Extra bright or bold text</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Half bright text</td>
</tr>
<tr>
<td>A_REVERSE</td>
<td>Reverse-video text</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>The best highlighting mode available</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underlined text</td>
</tr>
</tbody>
</table>

So, to display a reverse-video status line on the top line of the screen, you could code:

```python
stdscr.addstr(0, 0, "Current mode: Typing mode", curses.A_REVERSE)
stdscr.refresh()
```

The curses library also supports color on those terminals that provide it. The most common such terminal is probably the Linux console, followed by color xterms.
To use color, you must call the `start_color()` function soon after calling `initscr()`, to initialize the default color set (the `curses.wrapper()` function does this automatically). Once that’s done, the `has_colors()` function returns `TRUE` if the terminal in use can actually display color. (Note: curses uses the American spelling ‘color’, instead of the Canadian/British spelling ‘colour’. If you’re used to the British spelling, you’ll have to resign yourself to misspelling it for the sake of these functions.)

The curses library maintains a finite number of color pairs, containing a foreground (or text) color and a background color. You can get the attribute value corresponding to a color pair with the `color_pair()` function; this can be bitwise-OR’ed with other attributes such as `A_REVERSE`, but again, such combinations are not guaranteed to work on all terminals.

An example, which displays a line of text using color pair 1:

```python
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

As I said before, a color pair consists of a foreground and background color. The `init_pair(n, f, b)` function changes the definition of color pair `n`, to foreground color `f` and background color `b`. Color pair 0 is hard-wired to white on black, and cannot be changed.

Colors are numbered, and `start_color()` initializes 8 basic colors when it activates color mode. They are: 0:black, 1:red, 2:green, 3:yellow, 4:blue, 5:magenta, 6:cyan, and 7:white. The `curses` module defines named constants for each of these colors: `curses.COLOR_BLACK`, `curses.COLOR_RED`, and so forth.

Let’s put all this together. To change color 1 to red text on a white
background, you would call:

```python
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

When you change a color pair, any text already displayed using that color pair will change to the new colors. You can also display new text in this color with:

```python
stdscr.addstr(0, 0, "RED ALERT!", curses.color_pair(1))
```

Very fancy terminals can change the definitions of the actual colors to a given RGB value. This lets you change color 1, which is usually red, to purple or blue or any other color you like. Unfortunately, the Linux console doesn’t support this, so I’m unable to try it out, and can’t provide any examples. You can check if your terminal can do this by calling `can_change_color()`, which returns `True` if the capability is there. If you’re lucky enough to have such a talented terminal, consult your system’s man pages for more information.
User Input

The C curses library offers only very simple input mechanisms. Python’s `curses` module adds a basic text-input widget. (Other libraries such as Urwid have more extensive collections of widgets.)

There are two methods for getting input from a window:

- `getch()` refreshes the screen and then waits for the user to hit a key, displaying the key if `echo()` has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.
- `getkey()` does the same thing but converts the integer to a string. Individual characters are returned as 1-character strings, and special keys such as function keys return longer strings containing a key name such as `KEY_UP` or `^G`.

It’s possible to not wait for the user using the `nodelay()` window method. After `nodelay(True)`, `getch()` and `getkey()` for the window become non-blocking. To signal that no input is ready, `getch()` returns `curses.ERR` (a value of -1) and `getkey()` raises an exception. There’s also a `halfdelay()` function, which can be used to (in effect) set a timer on each `getch()`; if no input becomes available within a specified delay (measured in tenths of a second), `curses` raises an exception.

The `getch()` method returns an integer; if it’s between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAGE`, `curses.KEY_HOME`, or `curses.KEY_LEFT`. The main loop of your program may look something like this:
while True:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break  # Exit the while loop
    elif c == curses.KEY_HOME:
        x = y = 0

The **curses.ascii** module supplies ASCII class membership functions that take either integer or 1-character string arguments; these may be useful in writing more readable tests for such loops. It also supplies conversion functions that take either integer or 1-character-string arguments and return the same type. For example, **curses.ascii.ctrl()** returns the control character corresponding to its argument.

There's also a method to retrieve an entire string, **getstr()**. It isn't used very often, because its functionality is quite limited; the only editing keys available are the backspace key and the Enter key, which terminates the string. It can optionally be limited to a fixed number of characters.

curses.echo()  # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0, 0, 15)

The **curses.textpad** module supplies a text box that supports an Emacs-like set of keybindings. Various methods of the **Textbox** class support editing with input validation and gathering the edit results either with or without trailing spaces. Here's an example:
import curses
from curses.textpad import Textbox, rectangle

def main(stdscr):
    stdscr.addstr(0, 0, "Enter IM message: (hit Ctrl-G to send)"

    editwin = curses.newwin(5, 30, 2, 1)
    rectangle(stdscr, 1, 0, 1+5+1, 1+30+1)
    stdscr.refresh()

    box = Textbox(editwin)

    # Let the user edit until Ctrl-G is struck.
    box.edit()

    # Get resulting contents
    message = box.gather()
For More Information

This HOWTO doesn’t cover some advanced topics, such as reading the contents of the screen or capturing mouse events from an xterm instance, but the Python library page for the `curses` module is now reasonably complete. You should browse it next.

If you’re in doubt about the detailed behavior of the curses functions, consult the manual pages for your curses implementation, whether it’s ncurses or a proprietary Unix vendor’s. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and ACS_* characters available to you.

Because the curses API is so large, some functions aren’t supported in the Python interface. Often this isn’t because they’re difficult to implement, but because no one has needed them yet. Also, Python doesn’t yet support the menu library associated with ncurses. Patches adding support for these would be welcome; see the Python Developer’s Guide to learn more about submitting patches to Python.

- **Writing Programs with NCURSES**: a lengthy tutorial for C programmers.
- The `ncurses` man page
- The `ncurses` FAQ
- “Use curses... don’t swear”: video of a PyCon 2013 talk on controlling terminals using curses or Urwid.
- “Console Applications with Urwid”: video of a PyCon CA 2012 talk demonstrating some applications written using Urwid.
Descriptor HowTo Guide

Author: Raymond Hettinger
Contact: <python at rcn dot com>

Contents

- Descriptor HowTo Guide
  - Abstract
  - Definition and Introduction
  - Descriptor Protocol
  - Invoking Descriptors
  - Descriptor Example
  - Properties
  - Functions and Methods
  - Static Methods and Class Methods
Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in Python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.
Definition and Introduction

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are \texttt{__get__()}, \texttt{__set__()}, and \texttt{__delete__()}. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, \texttt{a.x} has a lookup chain starting with \texttt{a.__dict__['x']}, then \texttt{type(a).__dict__['x']}, and continuing through the base classes of \texttt{type(a)} excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and \texttt{super()}. They are used throughout Python itself to implement the new style classes introduced in version 2.2. Descriptors simplify the underlying C-code and offer a flexible set of new tools for everyday Python programs.
Descriptor Protocol

descr.__get__(self, obj, type=None) -> value

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines both __get__() and __set__(), it is considered a data descriptor. Descriptors that only define __get__() are called non-data descriptors (they are typically used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance’s dictionary. If an instance’s dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance’s dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

To make a read-only data descriptor, define both __get__() and __set__() with the __set__() raising an AttributeError when called. Defining the __set__() method with an exception raising placeholder is enough to make it a data descriptor.
Invoking Descriptors

A descriptor can be called directly by its method name. For example, d.__get__(obj).

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, obj.d looks up d in the dictionary of obj. If d defines the method __get__(), then d.__get__(obj) is invoked according to the precedence rules listed below.

The details of invocation depend on whether obj is an object or a class.

For objects, the machinery is in object.__getattribute__() which transforms b.x into type(b).__dict__['x'].__get__(b, type(b)). The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to __getattr__() if provided. The full C implementation can be found in PyObject_GenericGetAttr() in Objects/object.c.

For classes, the machinery is in type.__getattribute__() which transforms B.x into B.__dict__['x'].__get__(None, B). In pure Python, it looks like:

```python
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```
The important points to remember are:

- descriptors are invoked by the `__getattribute__()` method
- overriding `__getattribute__()` prevents automatic descriptor calls
- `object.__getattribute__() ` and `type.__getattribute__() ` make different calls to `__get__()`. 
- data descriptors always override instance dictionaries.
- non-data descriptors may be overridden by instance dictionaries.

The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The call `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately following `B` and then returns `A.__dict__['m'].__get__(obj, B)`. If not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

The implementation details are in `super_getattro()` in `Objects/typeobject.c` and a pure Python equivalent can be found in Guido's Tutorial.

The details above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`. Classes inherit this machinery when they derive from `object` or if they have a meta-class providing similar functionality. Likewise, classes can turn-off descriptor invocation by overriding `__getattribute__()`. 
Descriptor Example

The following code creates a class whose objects are data descriptors which print a message for each get or set. Overriding \_\_getattribute\_\_() is alternate approach that could do this for every attribute. However, this descriptor is useful for monitoring just a few chosen attributes:

class RevealAccess(object):
    """A data descriptor that sets and returns values normally and prints a message logging their access.
    """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...     ...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound methods, static methods, and class methods are all based on the descriptor protocol.
Properties

Calling `property()` is a succinct way of building a data descriptor that triggers function calls upon access to an attribute. Its signature is:

```python
property(fget=None, fset=None, fdel=None, doc=None)
```

The documentation shows a typical use to define a managed attribute `x`:

```python
class C(object):
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```python
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
```
The `property()` builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```python
class Cell(object):
    if self.fget is None:
        raise AttributeError("unreadable attribute")
    return self.fget(obj)

def __set__(self, obj, value):
    if self.fset is None:
        raise AttributeError("can't set attribute")
    self.fset(obj, value)

def __delete__(self, obj):
    if self.fdel is None:
        raise AttributeError("can't delete attribute")
    self.fdel(obj)

def getter(self, fget):
    return type(self)(fget, self.fset, self.fdel)

def setter(self, fset):
    return type(self)(self.fget, fset, self.fdel)

def deleter(self, fdel):
    return type(self)(self.fget, self.fset, fdel)
```
```python
def getvalue(self):
    "Recalculate the cell before returning value"
    self.recalc()
    return self._value

value = property(getvalue)
```
Functions and Methods

Python’s object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Class dictionaries store methods as functions. In a class definition, methods are written using `def` or `lambda`, the usual tools for creating functions. Methods only differ from regular functions in that the first argument is reserved for the object instance. By Python convention, the instance reference is called `self` but may be called `this` or any other variable name.

To support method calls, functions include the `__get__()` method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound methods when they are invoked from an object. In pure Python, it works like this:

```python
class Function(object):
    ...
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        if obj is None:
            return self
        return types.MethodType(self, obj)
```

Running the interpreter shows how the function descriptor works in practice:

```python
>>> class D(object):
...     def f(self, x):
...         return x
```
>>> d = D()

# Access through the class dictionary does not invoke
# It just returns the underlying function object.
>>> D.__dict__['f']
<function D.f at 0x00C45070>

# Dotted access from a class calls __get__() which just
# the underlying function unchanged.
>>> D.f
<function D.f at 0x00C45070>

# The function has a __qualname__ attribute to support
>>> D.f.__qualname__
'D.f'

# Dotted access from an instance calls __get__() which
# function wrapped in a bound method object
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>

# Internally, the bound method stores the underlying
# the bound instance, and the class of the bound instance.
>>> d.f.__func__
<function D.f at 0x1012e5ae8>
>>> d.f.__self__
<_main__.D object at 0x1012e1f98>
>>> d.f.__class__
<class 'method'>
Static Methods and Class Methods

Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

To recap, functions have a `__get__()` method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an `obj.f(*args)` call into `f(obj, *args)`. Calling `klass.f(*args)` becomes `f(*args)`.

This chart summarizes the binding and its two most useful variants:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Called from an Object</th>
<th>Called from a Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td><code>f(obj, *args)</code></td>
<td><code>f(*args)</code></td>
</tr>
<tr>
<td>staticmethod</td>
<td><code>f(*args)</code></td>
<td><code>f(*args)</code></td>
</tr>
<tr>
<td>classmethod</td>
<td><code>f(type(obj), *args)</code></td>
<td><code>f(klass, *args)</code></td>
</tr>
</tbody>
</table>

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattribute__(c, "f")` or `object.__getattribute__(C, "f")`. As a result, the function becomes identically accessible from either an object or a class.

Good candidates for static methods are methods that do not reference the `self` variable.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are
conceptually related but do not depend on the data. For instance, \( \text{erf}(x) \) is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: \( \text{s.erf}(1.5) \rightarrow 0.9332 \) or \( \text{Sample.erf}(1.5) \rightarrow 0.9332 \).

Since staticmethods return the underlying function with no changes, the example calls are unexciting:

```python
>>> class E(object):
...     def f(x):
...         print(x)
...     f = staticmethod(f)
... >>> print(E.f(3))
3
>>> print(E().f(3))
3
```

Using the non-data descriptor protocol, a pure Python version of `staticmethod()` would look like this:

```python
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=\None):
        return self.f
```

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:
```python
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f =classmethod(f)
... >>> print(E.f(3))
('E', 3)
>>> print(E().f(3))
('E', 3)
```

This behavior is useful whenever the function only needs to have a class reference and does not care about any underlying data. One use for classmethods is to create alternate class constructors. In Python 2.3, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```python
class Dict(object):
    ...
    def fromkeys(klass, iterable, value=None):
        """Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys =classmethod(fromkeys)
```

Now a new dictionary of unique keys can be constructed like this:

```python
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of `classmethod()` would look like this:
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
In this document, we'll take a tour of Python's features suitable for implementing programs in a functional style. After an introduction to the concepts of functional programming, we'll look at language features such as iterators and generator and relevant library modules such as itertools and functools.
Introduction

This section explains the basic concept of functional programming; if you’re just interested in learning about Python language features, skip to the next section on Iterators.

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**: programs are lists of instructions that tell the computer what to do with the program’s input. C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language you’re most likely to be familiar with; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.
- **Object-oriented** programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but don’t force the use of object-oriented features.
- **Functional** programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input. Well-known functional languages include the ML family (Standard ML, OCaml, and other variants) and Haskell.

The designers of some computer languages choose to emphasize one particular approach to programming. This often makes it difficult to
write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren’t visible in the function’s return value. Functions that have no side effects at all are called purely functional. Avoiding side effects means not using data structures that get updated as a program runs; every function’s output must only depend on its input.

Some languages are very strict about purity and don’t even have assignment statements such as \( a = 3 \) or \( c = a + b \), but it’s difficult to avoid all side effects. Printing to the screen or writing to a disk file are side effects, for example. For example, in Python a call to the \texttt{print()} or \texttt{time.sleep()} function both return no useful value; they’re only called for their side effects of sending some text to the screen or pausing execution for a second.

Python programs written in functional style usually won’t go to the extreme of avoiding all I/O or all assignments; instead, they’ll provide a functional-appearing interface but will use non-functional features internally. For example, the implementation of a function will still use assignments to local variables, but won’t modify global variables or have other side effects.

Functional programming can be considered the opposite of object-
oriented programming. Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes. Functional programming wants to avoid state changes as much as possible and works with data flowing between functions. In Python you might combine the two approaches by writing functions that take and return instances representing objects in your application (e-mail messages, transactions, etc.).

Functional design may seem like an odd constraint to work under. Why should you avoid objects and side effects? There are theoretical and practical advantages to the functional style:

- Formal provability.
- Modularity.
- Composability.
- Ease of debugging and testing.

**Formal provability**

A theoretical benefit is that it's easier to construct a mathematical proof that a functional program is correct.

For a long time researchers have been interested in finding ways to mathematically prove programs correct. This is different from testing a program on numerous inputs and concluding that its output is usually correct, or reading a program’s source code and concluding that the code looks right; the goal is instead a rigorous proof that a program produces the right result for all possible inputs.

The technique used to prove programs correct is to write down **invariants**, properties of the input data and of the program’s variables that are always true. For each line of code, you then show that if
invariants X and Y are true before the line is executed, the slightly different invariants X' and Y' are true after the line is executed. This continues until you reach the end of the program, at which point the invariants should match the desired conditions on the program’s output.

Functional programming’s avoidance of assignments arose because assignments are difficult to handle with this technique; assignments can break invariants that were true before the assignment without producing any new invariants that can be propagated onward.

Unfortunately, proving programs correct is largely impractical and not relevant to Python software. Even trivial programs require proofs that are several pages long; the proof of correctness for a moderately complicated program would be enormous, and few or none of the programs you use daily (the Python interpreter, your XML parser, your web browser) could be proven correct. Even if you wrote down or generated a proof, there would then be the question of verifying the proof; maybe there’s an error in it, and you wrongly believe you’ve proved the program correct.

Modularity

A more practical benefit of functional programming is that it forces you to break apart your problem into small pieces. Programs are more modular as a result. It’s easier to specify and write a small function that does one thing than a large function that performs a complicated transformation. Small functions are also easier to read and to check for errors.

Ease of debugging and testing
Testing and debugging a functional-style program is easier.

Debugging is simplified because functions are generally small and clearly specified. When a program doesn’t work, each function is an interface point where you can check that the data are correct. You can look at the intermediate inputs and outputs to quickly isolate the function that’s responsible for a bug.

Testing is easier because each function is a potential subject for a unit test. Functions don’t depend on system state that needs to be replicated before running a test; instead you only have to synthesize the right input and then check that the output matches expectations.

**Composability**

As you work on a functional-style program, you’ll write a number of functions with varying inputs and outputs. Some of these functions will be unavoidably specialized to a particular application, but others will be useful in a wide variety of programs. For example, a function that takes a directory path and returns all the XML files in the directory, or a function that takes a filename and returns its contents, can be applied to many different situations.

Over time you’ll form a personal library of utilities. Often you’ll assemble new programs by arranging existing functions in a new configuration and writing a few functions specialized for the current task.
Iterators

I’ll start by looking at a Python language feature that’s an important foundation for writing functional-style programs: iterators.

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called \texttt{__next__()} that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, \texttt{__next__()} must raise the \texttt{StopIteration} exception. Iterators don’t have to be finite, though; it’s perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in \texttt{iter()} function takes an arbitrary object and tries to return an iterator that will return the object’s contents or elements, raising \texttt{TypeError} if the object doesn’t support iteration. Several of Python’s built-in data types support iteration, the most common being lists and dictionaries. An object is called \texttt{iterable} if you can get an iterator for it.

You can experiment with the iteration interface manually:

```python
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it  #doctest: +ELLIPSIS
<...iterator object at ...>
>>> it.__next__()  # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
```

Python expects iterable objects in several different contexts, the most important being the `for` statement. In the statement `for X in Y`, `Y` must be an iterator or some object for which `iter()` can create an iterator. These two statements are equivalent:

```python
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

Iterators can be materialized as lists or tuples by using the `list()` or `tuple()` constructor functions:

```python
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequence unpacking also supports iterators: if you know an iterator will return N elements, you can unpack them into an N-tuple:

```python
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Built-in functions such as `max()` and `min()` can take a single iterator
argument and will return the largest or smallest element. The "in" and "not in" operators also support iterators: X in iterator is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite; max(), min() will never return, and if the element X never appears in the stream, the "in" and "not in" operators won't return either.

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the __next__() method. Functions may therefore consume all of the iterator's output, and if you need to do something different with the same stream, you'll have to create a new iterator.

Data Types That Support Iterators

We've already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling iter() on a dictionary returns an iterator that will loop over the dictionary's keys:

```python
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
```
Note that starting with Python 3.7, dictionary iteration order is guaranteed to be the same as the insertion order. In earlier versions, the behaviour was unspecified and could vary between implementations.

Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over values or key/value pairs, you can explicitly call the `values()` or `items()` methods to get an appropriate iterator.

The `dict()` constructor can accept an iterator that returns a finite stream of (key, value) tuples:

```python
given = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
dict(iter(given))
{'Italy': 'Rome', 'France': 'Paris', 'US': 'Washington DC'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this:

```python
for line in file:
    # do something for each line
...
```
Sets can take their contents from an iterable and let you iterate over the set's elements:

```python
S = {2, 3, 5, 7, 11, 13}
for i in S:
    print(i)
```
Generator expressions and list comprehensions

Two common operations on an iterator’s output are 1) performing some operation for every element, 2) selecting a subset of elements that meet some condition. For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions (short form: “listcomps” and “genexps”) are a concise notation for such operations, borrowed from the functional programming language Haskell (https://www.haskell.org/). You can strip all the whitespace from a stream of strings with the following code:

```python
line_list = [' line 1\n', 'line 2 \n', '...

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

You can select only certain elements by adding an "if" condition:

```python
stripped_list = [line.strip() for line in line_list if line != '']
```

With a list comprehension, you get back a Python list; `stripped_list` is a list containing the resulting lines, not an iterator. Generator expressions return an iterator that computes the values as necessary, not needing to materialize all the values at once. This
means that list comprehensions aren’t useful if you’re working with iterators that return an infinite stream or a very large amount of data. Generator expressions are preferable in these situations.

Generator expressions are surrounded by parentheses (“()”) and list comprehensions are surrounded by square brackets (“[]”). Generator expressions have the form:

```
( expression for expr in sequence1
  if condition1
  for expr2 in sequence2
    if condition2
  for expr3 in sequence3 ...
    if condition3
  for exprN in sequenceN
    if conditionN )
```

Again, for a list comprehension only the outside brackets are different (square brackets instead of parentheses).

The elements of the generated output will be the successive values of expression. The if clauses are all optional; if present, expression is only evaluated and added to the result when condition is true.

Generator expressions always have to be written inside parentheses, but the parentheses signalling a function call also count. If you want to create an iterator that will be immediately passed to a function you can write:

```
obj_total = sum(obj.count for obj in list_all_object
```

The for...in clauses contain the sequences to be iterated over.
The sequences do not have to be the same length, because they are iterated over from left to right, **not** in parallel. For each element in `sequence1`, `sequence2` is looped over from the beginning. `sequence3` is then looped over for each resulting pair of elements from `sequence1` and `sequence2`.

To put it another way, a list comprehension or generator expression is equivalent to the following Python code:

```python
for expr1 in sequence1:
    if not (condition1):
        continue  # Skip this element
for expr2 in sequence2:
    if not (condition2):
        continue  # Skip this element
...
for exprN in sequenceN:
    if not (conditionN):
        continue  # Skip this element

# Output the value of
# the expression.
```

This means that when there are multiple `for...in` clauses but no `if` clauses, the length of the resulting output will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```python
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2] #doctest:
[(('a', 1), ('a', 2), ('a', 3),
  ('b', 1), ('b', 2), ('b', 3),
  ('c', 1), ('c', 2), ('c', 3))]
```
To avoid introducing an ambiguity into Python’s grammar, if expression is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```python
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```
Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

You’re doubtless familiar with how regular function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the value is returned to the caller. A later call to the same function creates a new private namespace and a fresh set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```python
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
```

Any function containing a `yield` keyword is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn’t return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator’s state of execution is suspended and local variables are
preserved. On the next call to the generator’s `__next__()` method, the function will resume executing.

Here’s a sample usage of the `generate_ints()` generator:

```python
>>> gen = generate_ints(3)
>>> gen  #doctest: +ELLIPSIS
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Inside a generator function, `return` value causes `StopIteration(value)` to be raised from the `__next__()` method. Once this happens, or the bottom of the function is reached, the procession of values ends and the generator cannot yield any further values.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `__next__()` method increment `self.count` and return it. However, for a
moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python’s library, Lib/test/test_generators.py, contains a number of more interesting examples. Here’s one generator that implements an in-order traversal of a tree using generators recursively.

```python
# A recursive generator that generates Tree leaves in order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Two other examples in test_generators.py produce solutions for the N-Queens problem (placing N queens on an N\times N chess board so that no queen threatens another) and the Knight’s Tour (finding a route that takes a knight to every square of an N\times N chessboard without visiting any square twice).

**Passing values into a generator**

In Python 2.4 and earlier, generators only produced output. Once a generator’s code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. You could hack together this ability by making the generator look at a global variable or by passing in some mutable object that callers then modify, but these approaches are messy.
In Python 2.5 there's a simple way to pass values into a generator. `yield` became an expression, returning a value that can be assigned to a variable or otherwise operated on:

```python
val = (yield i)
```

I recommend that you **always** put parentheses around a `yield` expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(PEP 342 explains the exact rules, which are that a `yield-expression` must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator's code and the `yield` expression returns the specified value. If the regular `__next__()` method is called, the `yield` returns `None`.

Here's a simple counter that increments by 1 and allows changing the value of the internal counter.

```python
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
```
And here's an example of changing the counter:

```python
>>> it = counter(10)  # doctest: +SKIP
>>> next(it)  # doctest: +SKIP
0
>>> next(it)  # doctest: +SKIP
1
>>> it.send(8)  # doctest: +SKIP
8
>>> next(it)  # doctest: +SKIP
9
>>> next(it)  # doctest: +SKIP
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don’t just use its value in expressions unless you’re sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other methods on generators:

- `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator’s execution is paused.

- `close()` raises a `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator’s code must either raise `GeneratorExit` or
StopIteration; catching the exception and doing anything else is illegal and will trigger a RuntimeError. close() will also be called by Python's garbage collector when the generator is garbage-collected.

If you need to run cleanup code when a GeneratorExit occurs, I suggest using a try: ... finally: suite instead of catching GeneratorExit.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become coroutines, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a return statement), but coroutines can be entered, exited, and resumed at many different points (the yield statements).
Built-in functions

Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()` duplicate the features of generator expressions:

`map(f, iterA, iterB, ...)` returns an iterator over the sequence

\[
\text{f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...}
\]

```python
>>> def upper(s):
...     return s.upper()
```

```python
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
```

You can of course achieve the same effect with a list comprehension.

`filter(predicate, iter)` returns an iterator over all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

```python
>>> def is_even(x):
...     return (x % 2) == 0
```

```python
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```
This can also be written as a list comprehension:

```python
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` counts off the elements in the iterable returning 2-tuples containing the count (from `start`) and each element.

```python
>>> for item in enumerate(['subject', 'verb', 'object', ...      print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```python
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The `key` and `reverse` arguments are passed through to the constructed list's `sort()` method.

```python
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
```
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]

(For a more detailed discussion of sorting, see the Sorting HOW TO.)

The any(iter) and all(iter) built-ins look at the truth values of an iterable's contents. any() returns True if any element in the iterable is a true value, and all() returns True if all of the elements are true values:

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
>>> all([1, 1, 1])
True
```

zip(iterA, iterB, ...) takes one element from each iterable and returns them in a tuple:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
('a', 1), ('b', 2), ('c', 3)
```

It doesn't construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they're requested. (The technical term for this behaviour is lazy evaluation.)
This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

```python
zip(['a', 'b'], (1, 2, 3)) =>
('a', 1), ('b', 2)
```

You should avoid doing this, though, because an element may be taken from the longer iterators and discarded. This means you can’t go on to use the iterators further because you risk skipping a discarded element.
The `itertools` module

The `itertools` module contains a number of commonly-used iterators as well as functions for combining several iterators. This section will introduce the module’s contents by showing small examples.

The module’s functions fall into a few broad classes:

- Functions that create a new iterator based on an existing iterator.
- Functions for treating an iterator’s elements as function arguments.
- Functions for selecting portions of an iterator’s output.
- A function for grouping an iterator’s output.

Creating new iterators

`itertools.count(start, step)` returns an infinite stream of evenly spaced values. You can optionally supply the starting number, which defaults to 0, and the interval between numbers, which defaults to 1:

```python
itertools.count() =>
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
    10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.
```python
itertools.cycle([1, 2, 3, 4, 5]) => 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...

itertools.repeat(elem, [n]) returns the provided element n times, or returns the element endlessly if n is not provided.
```

```python
itertools.repeat('abc') => abc, abc, abc, abc, abc, abc, abc, abc, abc
itertools.repeat('abc', 5) => abc, abc, abc, abc, abc
```

```python
itertools.chain(iterA, iterB, ...) takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.
```

```python
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) => a, b, c, 1, 2, 3
```

```python
itertools.islice(iter, [start], stop, [step]) returns a stream that's a slice of the iterator. With a single stop argument, it will return the first stop elements. If you supply a starting index, you'll get stop-start elements, and if you supply a value for step, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for start, stop, or step.
```

```python
itertools.islice(range(10), 8) => 0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) => 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) => 2, 4, 6
```
**itertools.tee(iter, [n])** replicates an iterator; it returns \( n \) independent iterators that will all return the contents of the source iterator. If you don’t supply a value for \( n \), the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

```python
import itertools

# Call itertools.tee to create two independent iterators
iterA, iterB = itertools.tee(itertools.count(), 2)

# Use the iterators
for i in iterA:
    print(i)
for i in iterB:
    print(i)
```

Calling functions on elements

The **operator** module contains a set of functions corresponding to Python’s operators. Some examples are **operator.add(a, b)** (adds two values), **operator.ne(a, b)** (same as \( a \neq b \)), and **operator.attrgetter('id')** (returns a callable that fetches the `.id` attribute).

**itertools.starmap(func, iter)** assumes that the iterable will return a stream of tuples, and calls `func` using these tuples as the arguments:

```python
import os

# Call itertools.starmap to join paths
print(itertools.starmap(os.path.join,
                         [('/bin', 'python'), ('/usr', 'bin', 'python'), ('/usr', 'bin', 'java'), ('/usr', 'bin', 'perl'), ('/usr', 'bin', 'python')])
```

```bash
/bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/perl
```
Selecting elements

Another group of functions chooses a subset of an iterator’s elements based on a predicate.

`itertools.filterfalse(predicate, iter)` is the opposite of `filter()`, returning all elements for which the predicate returns false:

```
>>> itertools.filterfalse(is_even, itertools.count()) =>
1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```
def less_than_10(x):
    return x < 10

>>> itertools.takewhile(less_than_10, itertools.count())
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

>>> itertools.takewhile(is_even, itertools.count()) =>
0
```

`itertools.dropwhile(predicate, iter)` discards elements while the predicate returns true, and then returns the rest of the iterable's results.

```
>>> itertools.dropwhile(less_than_10, itertools.count())
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
```
```python
import itertools
dropwhile(is_even, itertools.count()) =>
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

`itertools.compress(data, selectors)` takes two iterators and returns only those elements of `data` for which the corresponding element of `selectors` is true, stopping whenever either one is exhausted:

```python
itertools.compress([1, 2, 3, 4, 5], [True, True, False])
1, 2, 5
```

**Combinatoric functions**

The `itertools.combinations(iterable, r)` returns an iterator giving all possible `r`-tuple combinations of the elements contained in `iterable`.

```python
itertools.combinations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 3), (2, 4), (2, 5),
(3, 4), (3, 5),
(4, 5)
```

```python
itertools.combinations([1, 2, 3, 4, 5], 3) =>
(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5),
(2, 3, 4), (2, 3, 5), (2, 4, 5),
(3, 4, 5)
```

The elements within each tuple remain in the same order as `iterable` returned them. For example, the number 1 is always before 2, 3, 4, or 5 in the examples above. A similar function,
**itertools.permutations(iterable, r=None)**, removes this constraint on the order, returning all possible arrangements of length r:

```python
itertools.permutations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 1), (2, 3), (2, 4), (2, 5),
(3, 1), (3, 2), (3, 4), (3, 5),
(4, 1), (4, 2), (4, 3), (4, 5),
(5, 1), (5, 2), (5, 3), (5, 4)
```

```python
itertools.permutations([1, 2, 3, 4, 5]) =>
(1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
...
(5, 4, 3, 2, 1)
```

If you don’t supply a value for r the length of the iterable is used, meaning that all the elements are permuted.

Note that these functions produce all of the possible combinations by position and don’t require that the contents of iterable are unique:

```python
itertools.permutations('aba', 3) =>
('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

The identical tuple ('a', 'a', 'b') occurs twice, but the two ‘a’ strings came from different positions.

The
**itertools.combinations_with_replacement(iterable, r)** function relaxes a different constraint: elements can be repeated within a single tuple. Conceptually an element is selected for the first position of each tuple and then is replaced before the second element is
selected.

```python
import itertools

combinations_with_replacement([1, 2, 3, 4,
    (1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
    (2, 2), (2, 3), (2, 4), (2, 5),
    (3, 3), (3, 4), (3, 5),
    (4, 4), (4, 5),
    (5, 5)]
```

**Grouping elements**

The last function I'll discuss, `itertools.groupby(iter, key_func=None)`, is the most complicated. `key_func(elem)` is a function that can compute a key value for each element returned by the iterable. If you don't supply a key function, the key is simply each element itself.

`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

```python
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'),
            ('Anchorage', 'AK'), ('Nome', 'AK'),
            ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'),
            ...
]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
    ('AL', iterator-1),
    ('AK', iterator-2),
    ('AZ', iterator-3), ...
```
where
iterator-1 =>
  ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma',
iterator-2 =>
  ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
  ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson',

`groupby()` assumes that the underlying iterable’s contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.
The functions module

The `functools` module in Python 2.5 contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c);` you may wish to create a new function `g(b, c)` that's equivalent to `f(1, b, c);` you're filling in a value for one of `f()`'s parameters. This is called “partial function application”.

The constructor for `partial()` takes the arguments `(function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```python
import functools
def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."
    print('%%s: %%s' %% (subsystem, message))
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

```python
functools.reduce(func, iter, [initial_value])
```
cumulatively performs an operation on all the iterable's elements and, therefore, can't be applied to infinite iterables. `func` must be a function that takes two elements and returns a single value. `functools.reduce()` takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a `TypeError` exception is raised. If the initial value is supplied, it's used as a starting point and `func(initial_value, A)` is the first calculation.

```python
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C', 'ABBC'])
Traceback (most recent call last):
 ... error: reduce() of empty sequence with no initial value
```

If you use `operator.add()` with `functools.reduce()`, you'll add up all the elements of the iterable. This case is so common that there's a special built-in called `sum()` to compute it:

```python
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([[]])
```

For many uses of `functools.reduce()`, though, it can be clearer to just write the obvious `for` loop:

```python
import functools

# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3],

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

A related function is `itertools.accumulate(iterable, func=operator.add)`. It performs the same calculation, but instead of returning only the final result, `accumulate()` returns an iterator that also yields each partial result:

```python
itertools.accumulate([1, 2, 3, 4, 5]) => 1, 3, 6, 10, 15

itertools.accumulate([1, 2, 3, 4, 5], operator.mul) 1, 2, 6, 24, 120
```

**The operator module**

The `operator` module was mentioned earlier. It contains a set of functions corresponding to Python's operators. These functions are often useful in functional-style code because they save you from writing trivial functions that perform a single operation.
Some of the functions in this module are:

- Math operations: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- Logical operations: `not_()`, `truth()`.
- Bitwise operations: `and_()`, `or_()`, `invert()`.
- Comparisons: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, and `ge()`.
- Object identity: `is_()`, `is_not()`.

Consult the operator module's documentation for a complete list.
Small functions and the lambda expression

When writing functional-style programs, you'll often need little functions that act as predicates or that combine elements in some way.

If there's a Python built-in or a module function that's suitable, you don't need to define a new function at all:

```python
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn't exist, you need to write it. One way to write small functions is to use the `lambda` expression. `lambda` takes a number of parameters and an expression combining these parameters, and creates an anonymous function that returns the value of the expression:

```python
adder = lambda x, y: x + y
print_assign = lambda name, value: name + '=' + str(value)
```

An alternative is to just use the `def` statement and define a function in the usual way:

```python
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

Which alternative is preferable? That's a style question; my usual
course is to avoid using \texttt{lambda}.

One reason for my preference is that \texttt{lambda} is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway \texttt{if}... \texttt{elif}... \texttt{else} comparisons or \texttt{try}... \texttt{except} statements. If you try to do too much in a \texttt{lambda} statement, you'll end up with an overly complicated expression that's hard to read. Quick, what's the following code doing?

```python
import functools
total = functools.reduce(\texttt{lambda} \ a, \ b: (0, \ a[1] + \ b[1]), \ items)[1]
```

You can figure it out, but it takes time to disentangle the expression to figure out what's going on. Using a short nested \texttt{def} statements makes things a little bit better:

```python
import functools
def combine(a, b):
    \textbf{return} 0, a[1] + b[1]

total = functools.reduce(combine, \ items)[1]
```

But it would be best of all if I had simply used a \texttt{for} loop:

```python
total = 0
\textbf{for} a, b \ \textbf{in} \ \texttt{items}:
    total +-= b
```

Or the \texttt{sum()} built-in and a generator expression:

```python
total = \texttt{sum}(b \ \textbf{for} \ a, b \ \textbf{in} \ \texttt{items})
```
Many uses of `functools.reduce()` are clearer when written as for loops.

Fredrik Lundh once suggested the following set of rules for refactoring uses of lambda:

1. Write a lambda function.
2. Write a comment explaining what the heck that lambda does.
3. Study the comment for a while, and think of a name that captures the essence of the comment.
4. Convert the lambda to a def statement, using that name.
5. Remove the comment.

I really like these rules, but you’re free to disagree about whether this lambda-free style is better.
Revision History and Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged genexp and listcomp sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the functional module written by Collin Winter; adds short section on the operator module; a few other edits.
References

General

**Structure and Interpretation of Computer Programs**, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. Full text at [https://mitpress.mit.edu/sicp/](https://mitpress.mit.edu/sicp/). In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

[http://www.defmacro.org/ramblings/fp.html](http://www.defmacro.org/ramblings/fp.html): A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.


Python-specific


Mertz also wrote a 3-part series of articles on functional programming for IBM’s DeveloperWorks site; see [part 1](https://www.ibm.com/developerworks/), [part 2](https://www.ibm.com/developerworks/), and [part 3](https://www.ibm.com/developerworks/).
Python documentation

Documentation for the **itertools** module.

Documentation for the **functools** module.

Documentation for the **operator** module.

**PEP 289**: “Generator Expressions”

**PEP 342**: “Coroutines via Enhanced Generators” describes the new generator features in Python 2.5.
Basic Logging Tutorial

Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

When to use logging

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

<table>
<thead>
<tr>
<th>Task you want to perform</th>
<th>The best tool for the task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display console output for ordinary usage of a command line script or program</td>
<td><code>print()</code></td>
</tr>
<tr>
<td>Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)</td>
<td><code>logging.info()</code> (or <code>logging.debug()</code> for very detailed output for diagnostic purposes)</td>
</tr>
<tr>
<td></td>
<td><code>warnings.warn()</code> in library code if the issue is avoidable and the client application should be</td>
</tr>
</tbody>
</table>
Issue a warning regarding a particular runtime event

<table>
<thead>
<tr>
<th>Level</th>
<th>When it’s used</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>Detailed information, typically of interest only when diagnosing problems.</td>
</tr>
<tr>
<td>INFO</td>
<td>Confirmation that things are working as expected.</td>
</tr>
<tr>
<td>WARNING</td>
<td>An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Due to a more serious problem, the software has not been able to perform some function.</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>A serious error, indicating that the program itself may be unable to continue running.</td>
</tr>
</tbody>
</table>

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Report an error regarding a particular runtime event

- **logging.error()**, **logging.exception()** or **logging.critical()** as appropriate for the specific error and application domain

Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)

- **logging.warning()** if there is nothing the client application can do about the situation, but the event should still be noted

Raise an exception
The default level is `WARNING`, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.

### A simple example

A very simple example is:

```python
import logging
logging.warning('Watch out!')  # will print a message
logging.info('I told you so')  # will not print anything
```

If you type these lines into a script and run it, you’ll see:

```console
WARNING:root:Watch out!
```

printed out on the console. The `INFO` message doesn’t appear because the default level is `WARNING`. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. ‘Watch out!’. Don’t worry about the ‘root’ part for now: it will be explained later. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

### Logging to a file

A very common situation is that of recording logging events in a file, so let’s look at that next. Be sure to try the following in a newly-started Python interpreter, and don’t just continue from the session described
import logging
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')

And now if we open the file and look at what we have, we should find the log messages:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to DEBUG, all of the messages were printed.

If you want to set the logging level from a command-line option such as:

```
--log=INFO
```

and you have the value of the parameter passed for `--log` in some variable `loglevel`, you can use:

```
getattr(logging, loglevel.upper())
```

to get the value which you'll pass to `basicConfig()` via the `level` argument. You may want to error check any user input value, perhaps as in the following example:
The call to `basicConfig()` should come before any calls to `debug()`, `info()` etc. As it's intended as a one-off simple configuration facility, only the first call will actually do anything: subsequent calls are effectively no-ops.

If you run the above script several times, the messages from successive runs are appended to the file `example.log`. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the `filemode` argument, by changing the call in the above example to:

```python
logging.basicConfig(filename='example.log', filemode=...)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

**Logging from multiple modules**

If your program consists of multiple modules, here's an example of how you could organize logging in it:

```python
# myapp.py
import logging
import mylib
```
```python
def main():
    logging.basicConfig(filename='myapp.log', level=logging.info('Started'))
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

```python
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

If you run `myapp.py`, you should see this in `myapp.log`:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

which is hopefully what you were expecting to see. You can generalize this to multiple modules, using the pattern in `mylib.py`. Note that for this simple usage pattern, you won’t know, by looking in the log file, where in your application your messages came from, apart from looking at the event description. If you want to track the location of your messages, you’ll need to refer to the documentation beyond the tutorial level – see Advanced Logging Tutorial.

### Logging variable data

To log variable data, use a format string for the event description
message and append the variable data as arguments. For example:

```python
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

```
WARNING:root:Look before you leap!
```

As you can see, merging of variable data into the event description message uses the old, %-style of string formatting. This is for backwards compatibility: the logging package pre-dates newer formatting options such as `str.format()` and `string.Template`. These newer formatting options are supported, but exploring them is outside the scope of this tutorial: see [Using particular formatting styles throughout your application](#) for more information.

### Changing the format of displayed messages

To change the format which is used to display messages, you need to specify the format you want to use:

```python
import logging
logging.basicConfig(format='%(levelname)s:%(message)s')
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

which would print:

```
DEBUG:This message should appear on the console
INFO:So should this
```
Notice that the ‘root’ which appeared in earlier examples has disappeared. For a full set of things that can appear in format strings, you can refer to the documentation for LogRecord attributes, but for simple usage, you just need thelevelname (severity), message (event description, including variable data) and perhaps to display when the event occurred. This is described in the next section.

Displaying the date/time in messages

To display the date and time of an event, you would place ‘%%(asctime)s’ in your format string:

```python
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

which should print something like this:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

The default format for date/time display (shown above) is like ISO8601 or RFC 3339. If you need more control over the formatting of the date/time, provide a datefmt argument to basicConfig, as in this example:

```python
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```
which would display something like this:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

The format of the `datefmt` argument is the same as supported by `time.strftime()`.

**Next Steps**

That concludes the basic tutorial. It should be enough to get you up and running with logging. There’s a lot more that the logging package offers, but to get the best out of it, you’ll need to invest a little more of your time in reading the following sections. If you’re ready for that, grab some of your favourite beverage and carry on.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don’t understand something, please post a question on the comp.lang.python Usenet group (available at https://groups.google.com/forum/#!forum/comp.lang.python) and you should receive help before too long.

Still here? You can carry on reading the next few sections, which provide a slightly more advanced/in-depth tutorial than the basic one above. After that, you can take a look at the Logging Cookbook.
The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Log event information is passed between loggers, handlers, filters and formatters in a `LogRecord` instance.

Logging is performed by calling methods on instances of the `Logger` class (hereafter called `loggers`). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named ‘scan’ is the parent of loggers ‘scan.text’, ‘scan.html’ and ‘scan.pdf’. Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

A good convention to use when naming loggers is to use a module-level logger, in each module which uses logging, named as follows:

```python
logger = logging.getLogger(__name__)
```

This means that logger names track the package/module hierarchy, and it’s intuitively obvious where events are logged just from the logger name.

The root of the hierarchy of loggers is called the root logger. That’s the
logger used by the functions `debug()`, `info()`, `warning()`, `error()` and `critical()`, which just call the same-named method of the root logger. The functions and the methods have the same signatures. The root logger’s name is printed as ‘root’ in the logged output.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, queues, or OS-specific logging mechanisms such as syslog or the Windows NT event log. Destinations are served by `handler` classes. You can create your own log destination class if you have special requirements not met by any of the built-in handler classes.

By default, no destination is set for any logging messages. You can specify a destination (such as console or file) by using `basicConfig()` as in the tutorial examples. If you call the functions `debug()`, `info()`, `warning()`, `error()` and `critical()`, they will check to see if no destination is set; and if one is not set, they will set a destination of the console (`sys.stderr`) and a default format for the displayed message before delegating to the root logger to do the actual message output.

The default format set by `basicConfig()` for messages is:

```
severity:logger name:message
```

You can change this by passing a format string to `basicConfig()` with the `format` keyword argument. For all options regarding how a format string is constructed, see `Formatter Objects`.

**Logging Flow**
The flow of log event information in loggers and handlers is illustrated in the following diagram.

Loggers

Logger objects have a threefold job. First, they expose several methods to application code so that applications can log messages at runtime. Second, logger objects determine which log messages to act upon based upon severity (the default filtering facility) or filter objects. Third, logger objects pass along relevant log messages to all interested log handlers.

The most widely used methods on logger objects fall into two categories: configuration and message sending.
These are the most common configuration methods:

- **Logger.setLevel()** specifies the lowest-severity log message a logger will handle, where debug is the lowest built-in severity level and critical is the highest built-in severity. For example, if the severity level is INFO, the logger will handle only INFO, WARNING, ERROR, and CRITICAL messages and will ignore DEBUG messages.
- **Logger.addHandler()** and **Logger.removeHandler()** add and remove handler objects from the logger object. Handlers are covered in more detail in Handlers.
- **Logger.addFilter()** and **Logger.removeFilter()** add and remove filter objects from the logger object. Filters are covered in more detail in Filter Objects.

You don’t need to always call these methods on every logger you create. See the last two paragraphs in this section.

With the logger object configured, the following methods create log messages:

- **Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), and Logger.critical()** all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of %s, %d, %f, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to **kwargs, the logging methods care only about a keyword of exc_info and use it to determine whether to log exception information.
- **Logger.exception()** creates a log message similar to Logger.error(). The difference is that Logger.exception() dumps a stack trace along with it. Call
this method only from an exception handler.

- **Logger.log()** takes a log level as an explicit argument. This is a little more verbose for logging messages than using the log level convenience methods listed above, but this is how to log at custom log levels.

**getLogger()** returns a reference to a logger instance with the specified name if it is provided, or **root** if not. The names are period-separated hierarchical structures. Multiple calls to **getLogger()** with the same name will return a reference to the same logger object. Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of **foo**, loggers with names of **foo.bar**, **foo.bar.baz**, and **foo.bam** are all descendants of **foo**.

Loggers have a concept of **effective level**. If a level is not explicitly set on a logger, the level of its parent is used instead as its effective level. If the parent has no explicit level set, its parent is examined, and so on - all ancestors are searched until an explicitly set level is found. The root logger always has an explicit level set (**WARNING** by default). When deciding whether to process an event, the effective level of the logger is used to determine whether the event is passed to the logger’s handlers.

Child loggers propagate messages up to the handlers associated with their ancestor loggers. Because of this, it is unnecessary to define and configure handlers for all the loggers an application uses. It is sufficient to configure handlers for a top-level logger and create child loggers as needed. (You can, however, turn off propagation by setting the **propagate** attribute of a logger to False.)

**Handlers**
**Handler** objects are responsible for dispatching the appropriate log messages (based on the log messages' severity) to the handler's specified destination. **Logger** objects can add zero or more handler objects to themselves with an `addHandler()` method. As an example scenario, an application may want to send all log messages to a log file, all log messages of error or higher to stdout, and all messages of critical to an email address. This scenario requires three individual handlers where each handler is responsible for sending messages of a specific severity to a specific location.

The standard library includes quite a few handler types (see Useful Handlers); the tutorials use mainly **StreamHandler** and **FileHandler** in its examples.

There are very few methods in a handler for application developers to concern themselves with. The only handler methods that seem relevant for application developers who are using the built-in handler objects (that is, not creating custom handlers) are the following configuration methods:

- The `setLevel()` method, just as in logger objects, specifies the lowest severity that will be dispatched to the appropriate destination. Why are there two `setLevel()` methods? The level set in the logger determines which severity of messages it will pass to its handlers. The level set in each handler determines which messages that handler will send on.
- `setFormatter()` selects a Formatter object for this handler to use.
- `addFilter()` and `removeFilter()` respectively configure and deconfigure filter objects on handlers.

Application code should not directly instantiate and use instances of **Handler**. Instead, the **Handler** class is a base class that defines the
interface that all handlers should have and establishes some default behavior that child classes can use (or override).

Formatters

Formatter objects configure the final order, structure, and contents of the log message. Unlike the base `logging.Handler` class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes three optional arguments – a message format string, a date format string and a style indicator.

```python
logging.Formatter.__init__(fmt=None, datefmt=None, style='%')
```

If there is no message format string, the default is to use the raw message. If there is no date format string, the default date format is:

```
%Y-%m-%d %H:%M:%S
```

with the milliseconds tacked on at the end. The style is one of %, '{' or '$'. If one of these is not specified, then '%' will be used.

If the style is ‘%’, the message format string uses %(<dictionary key>)s styled string substitution; the possible keys are documented in LogRecord attributes. If the style is ‘{’, the message format string is assumed to be compatible with `str.format()` (using keyword arguments), while if the style is ‘$’ then the message format string should conform to what is expected by `string.Template.substitute()`.

*Changed in version 3.2:* Added the style parameter.
The following message format string will log the time in a human-readable format, the severity of the message, and the contents of the message, in that order:

'%asctime)s - %levelname)s - %message)s'

Formatters use a user-configurable function to convert the creation time of a record to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute of the instance to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the Formatter class (to `time.gmtime` for GMT display).

Configuring Logging

Programmers can configure logging in three ways:

1. Creating loggers, handlers, and formatters explicitly using Python code that calls the configuration methods listed above.
2. Creating a logging config file and reading it using the `fileConfig()` function.
3. Creating a dictionary of configuration information and passing it to the `dictConfig()` function.

For the reference documentation on the last two options, see Configuration functions. The following example configures a very simple logger, a console handler, and a simple formatter using Python code:

```python
import logging
```
# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')

Running this module from the command line produces the following output:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

The following Python module creates a logger, handler, and formatter
nearly identical to those in the example listed above, with the only difference being the names of the objects:

```python
import logging
import logging.config
logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

Here is the logging.conf file:

```conf
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
```
The output is nearly identical to that of the non-config-file-based example:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

You can see that the config file approach has a few advantages over the Python code approach, mainly separation of configuration and code and the ability of noncoders to easily modify the logging properties.

**Warning:** The `fileConfig()` function takes a default parameter, `disable_existing_loggers`, which defaults to `True` for reasons of backward compatibility. This may or may not be what you want, since it will cause any non-root loggers existing before the `fileConfig()` call to be disabled unless they (or an ancestor) are
explicitly named in the configuration. Please refer to the reference
documentation for more information, and specify False for this
parameter if you wish.

The dictionary passed to `dictConfig()` can also specify a
Boolean value with key `disable_existing_loggers`, which if
not specified explicitly in the dictionary also defaults to being
interpreted as True. This leads to the logger-disabling behaviour
described above, which may not be what you want - in which case,
provide the key explicitly with a value of False.

Note that the class names referenced in config files need to be either
relative to the logging module, or absolute values which can be
resolved using normal import mechanisms. Thus, you could use either
`WatchedFileHandler` (relative to the logging module) or
`mypackage.mymodule.MyHandler` (for a class defined in package
`mypackage` and module `mymodule`, where `mypackage` is available
on the Python import path).

In Python 3.2, a new means of configuring logging has been
introduced, using dictionaries to hold configuration information. This
provides a superset of the functionality of the config-file-based
approach outlined above, and is the recommended configuration
method for new applications and deployments. Because a Python
dictionary is used to hold configuration information, and since you can
populate that dictionary using different means, you have more options
for configuration. For example, you can use a configuration file in
JSON format, or, if you have access to YAML processing functionality,
a file in YAML format, to populate the configuration dictionary. Or, of
course, you can construct the dictionary in Python code, receive it in
pickled form over a socket, or use whatever approach makes sense
for your application.

Here's an example of the same configuration as above, in YAML format for the new dictionary-based approach:

```yaml
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s

handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout

loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no

root:
  level: DEBUG
  handlers: [console]
```

For more information about logging using a dictionary, see Configuration functions.

What happens if no configuration is provided

If no logging configuration is provided, it is possible to have a situation where a logging event needs to be output, but no handlers can be found to output the event. The behaviour of the logging package in these circumstances is dependent on the Python version.
For versions of Python prior to 3.2, the behaviour is as follows:

- If `logging.raiseExceptions` is `False` (production mode), the event is silently dropped.
- If `logging.raiseExceptions` is `True` (development mode), a message ‘No handlers could be found for logger X.Y.Z’ is printed once.

In Python 3.2 and later, the behaviour is as follows:

- The event is output using a ‘handler of last resort’, stored in `logging.lastResort`. This internal handler is not associated with any logger, and acts like a `StreamHandler` which writes the event description message to the current value of `sys.stderr` (therefore respecting any redirections which may be in effect). No formatting is done on the message - just the bare event description message is printed. The handler’s level is set to `WARNING`, so all events at this and greater severities will be output.

To obtain the pre-3.2 behaviour, `logging.lastResort` can be set to `None`.

**Configuring Logging for a Library**

When developing a library which uses logging, you should take care to document how the library uses logging - for example, the names of loggers used. Some consideration also needs to be given to its logging configuration. If the using application does not use logging, and library code makes logging calls, then (as described in the previous section) events of severity `WARNING` and greater will be printed to `sys.stderr`. This is regarded as the best default behaviour.
If for some reason you don’t want these messages printed in the absence of any logging configuration, you can attach a do-nothing handler to the top-level logger for your library. This avoids the message being printed, since a handler will always be found for the library’s events: it just doesn’t produce any output. If the library user configures logging for application use, presumably that configuration will add some handlers, and if levels are suitably configured then logging calls made in library code will send output to those handlers, as normal.

A do-nothing handler is included in the logging package: `NullHandler` (since Python 3.1). An instance of this handler could be added to the top-level logger of the logging namespace used by the library (if you want to prevent your library’s logged events being output to `sys.stderr` in the absence of logging configuration). If all logging by a library `foo` is done using loggers with names matching ‘foo.x’, ‘foo.x.y’, etc. then the code:

```python
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

should have the desired effect. If an organisation produces a number of libraries, then the logger name specified can be ‘orgname.foo’ rather than just ‘foo’.

**Note:** It is strongly advised that you do not add any handlers other than `NullHandler` to your library’s loggers. This is because the configuration of handlers is the prerogative of the application developer who uses your library. The application developer knows their target audience and what handlers are most appropriate for their application: if you add handlers ‘under the hood’, you might well interfere with their ability to carry out unit tests and deliver logs which
suit their requirements.
Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
</tr>
<tr>
<td>NOTSET</td>
<td>0</td>
</tr>
</tbody>
</table>

Levels can also be associated with loggers, being set either by the developer or through loading a saved logging configuration. When a logging method is called on a logger, the logger compares its own level with the level associated with the method call. If the logger’s level is higher than the method call’s, no logging message is actually generated. This is the basic mechanism controlling the verbosity of logging output.

Logging messages are encoded as instances of the LogRecord class. When a logger decides to actually log an event, a LogRecord instance is created from the logging message.

Logging messages are subjected to a dispatch mechanism through
the use of *handlers*, which are instances of subclasses of the *Handler* class. Handlers are responsible for ensuring that a logged message (in the form of a *LogRecord*) ends up in a particular location (or set of locations) which is useful for the target audience for that message (such as end users, support desk staff, system administrators, developers). Handlers are passed *LogRecord* instances intended for particular destinations. Each logger can have zero, one or more handlers associated with it (via the *addHandler()* method of *Logger*). In addition to any handlers directly associated with a logger, *all handlers associated with all ancestors of the logger* are called to dispatch the message (unless the *propagate* flag for a logger is set to a false value, at which point the passing to ancestor handlers stops).

Just as for loggers, handlers can have levels associated with them. A handler’s level acts as a filter in the same way as a logger’s level does. If a handler decides to actually dispatch an event, the *emit()* method is used to send the message to its destination. Most user-defined subclasses of *Handler* will need to override this *emit()*.

**Custom Levels**

Defining your own levels is possible, but should not be necessary, as the existing levels have been chosen on the basis of practical experience. However, if you are convinced that you need custom levels, great care should be exercised when doing this, and it is possibly a *very bad idea to define custom levels if you are developing a library*. That’s because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean
different things for different libraries.
Useful Handlers

In addition to the base Handler class, many useful subclasses are provided:

1. StreamHandler instances send messages to streams (file-like objects).
2. FileHandler instances send messages to disk files.
3. BaseRotatingHandler is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use RotatingFileHandler or TimedRotatingFileHandler.
4. RotatingFileHandler instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. TimedRotatingFileHandler instances send messages to disk files, rotating the log file at certain timed intervals.
6. SocketHandler instances send messages to TCP/IP sockets. Since 3.4, Unix domain sockets are also supported.
7. DatagramHandler instances send messages to UDP sockets. Since 3.4, Unix domain sockets are also supported.
8. SMTPHandler instances send messages to a designated email address.
9. SysLogHandler instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. NTEventLogHandler instances send messages to a Windows NT/2000/XP event log.
11. MemoryHandler instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. HTTPHandler instances send messages to an HTTP server using either GET or POST semantics.
13. WatchedFileHandler instances watch the file they are logging to. If the file changes, it is closed and reopened using the file
name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.

14. **QueueHandler** instances send messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

15. **NullHandler** instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the ‘No handlers could be found for logger XXX’ message which can be displayed if the library user has not configured logging. See Configuring Logging for a Library for more information.

*New in version 3.1:* The **NullHandler** class.

*New in version 3.2:* The **QueueHandler** class.

The **NullHandler**, **StreamHandler** and **FileHandler** classes are defined in the core logging package. The other handlers are defined in a sub-module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Logged messages are formatted for presentation through instances of the **Formatter** class. They are initialized with a format string suitable for use with the % operator and a dictionary.

For formatting multiple messages in a batch, instances of **BufferingFormatter** can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough, instances of **Filter** can be added to both **Logger** and **Handler** instances (through their `addFilter()` method). Before
deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic **Filter** functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.
Exceptions raised during logging

The logging package is designed to swallow exceptions which occur while logging in production. This is so that errors which occur while handling logging events - such as logging misconfiguration, network or other similar errors - do not cause the application using logging to terminate prematurely.

SystemExit and KeyboardInterrupt exceptions are never swallowed. Other exceptions which occur during the emit() method of a Handler subclass are passed to its handleError() method.

The default implementation of handleError() in Handler checks to see if a module-level variable, raiseExceptions, is set. If set, a traceback is printed to sys.stderr. If not set, the exception is swallowed.

Note: The default value of raiseExceptions is True. This is because during development, you typically want to be notified of any exceptions that occur. It's advised that you set raiseExceptions to False for production usage.
Using arbitrary objects as messages

In the preceding sections and examples, it has been assumed that the message passed when logging the event is a string. However, this is not the only possibility. You can pass an arbitrary object as a message, and its `__str__()`, method will be called when the logging system needs to convert it to a string representation. In fact, if you want to, you can avoid computing a string representation altogether - for example, the `SocketHandler` emits an event by pickling it and sending it over the wire.
Optimization

Formatting of message arguments is deferred until it cannot be avoided. However, computing the arguments passed to the logging method can also be expensive, and you may want to avoid doing it if the logger will just throw away your event. To decide what to do, you can call the `isEnabledFor()` method which takes a level argument and returns true if the event would be created by the Logger for that level of call. You can write code like this:

```python
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1, expensive_func2)
```

so that if the logger's threshold is set above `DEBUG`, the calls to `expensive_func1()` and `expensive_func2()` are never made.

**Note:** In some cases, `isEnabledFor()` can itself be more expensive than you'd like (e.g. for deeply nested loggers where an explicit level is only set high up in the logger hierarchy). In such cases (or if you want to avoid calling a method in tight loops), you can cache the result of a call to `isEnabledFor()` in a local or instance variable, and use that instead of calling the method each time. Such a cached value would only need to be recomputed when the logging configuration changes dynamically while the application is running (which is not all that common).

There are other optimizations which can be made for specific applications which need more precise control over what logging information is collected. Here's a list of things you can do to avoid
processing during logging which you don’t need:

<table>
<thead>
<tr>
<th>What you don’t want to collect</th>
<th>How to avoid collecting it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information about where calls were made from.</td>
<td>Set <code>logging._srcfile</code> to <code>None</code>. This avoids calling <code>sys._getframe()</code>, which may help to speed up your code in environments like PyPy (which can’t speed up code that uses <code>sys._getframe()</code>), if and when PyPy supports Python 3.x.</td>
</tr>
<tr>
<td>Threading information.</td>
<td>Set <code>logging.logThreads</code> to <code>0</code>.</td>
</tr>
<tr>
<td>Process information.</td>
<td>Set <code>logging.logProcesses</code> to <code>0</code>.</td>
</tr>
</tbody>
</table>

Also note that the core logging module only includes the basic handlers. If you don’t import `logging.handlers` and `logging.config`, they won’t take up any memory.

See also:

**Module** [logging](#)  
API reference for the logging module.

**Module** [logging.config](#)  
Configuration API for the logging module.

**Module** [logging.handlers](#)  
Useful handlers included with the logging module.
This page contains a number of recipes related to logging, which have been found useful in the past.
Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```python
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)

# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)

# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)

# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s: %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)

# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
```

Here is the auxiliary module:

```python
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary
        self.logger.info('creating an instance of Auxiliary

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

    def some_function():
        module_logger.info('received a call to "some_function"
```

The output looks like this:

```
2005-03-23 23:47:11,665 - spam_application.auxiliary creating an instance of Auxiliary
```
2005-03-23 23:47:11,671 - spam_application - INFO - calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO - received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO - done with auxiliary_module.some_function()
Logging from multiple threads

Logging from multiple threads requires no special effort. The following example shows logging from the main (initial) thread and another thread:

```python
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format=info = {'stop': False}
    thread = threading.Thread(target=worker, args=(i
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

When run, the script should print something like the following:
This shows the logging output interspersed as one might expect. This approach works for more threads than shown here, of course.
Multiple handlers and formatters

Loggers are plain Python objects. The `addHandler()` method has no minimum or maximum quota for the number of handlers you may add. Sometimes it will be beneficial for an application to log all messages of all severities to a text file while simultaneously logging errors or above to the console. To set this up, simply configure the appropriate handlers. The logging calls in the application code will remain unchanged. Here is a slight modification to the previous simple module-based configuration example:

```python
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
```
Notice that the ‘application’ code does not care about multiple handlers. All that changed was the addition and configuration of a new handler named `fh`.

The ability to create new handlers with higher- or lower-severity filters can be very helpful when writing and testing an application. Instead of using many `print` statements for debugging, use `logger.debug`: Unlike the print statements, which you will have to delete or comment out later, the `logger.debug` statements can remain intact in the source code and remain dormant until you need them again. At that time, the only change that needs to happen is to modify the severity level of the `logger` and/or handler to debug.
Logging to multiple destinations

Let’s say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let’s also assume that the file should contain timestamps, but the console messages should not. Here’s how you can achieve this:

```python
import logging

# set up logging to file - see previous section for
logging.basicConfig(level=logging.DEBUG,
    format='%(asctime)s %(name)-12s
    datefmt=%m-%d %H:%M',
    filename='/temp/myapp.log',
    filemode='w')

# define a Handler which writes INFO messages or higher
c = logging.StreamHandler()
c.setLevel(logging.INFO)

# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
    # tell the handler to use this format
c.setFormatter(formatter)

# add the handler to the root logger
logging.getLogger('').addHandler(c)

# Now, we can log to the root logger, or any other logger
logging.info('Jackdaws love my big sphinx of quartz.

# Now, define a couple of other loggers which might
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')"
When you run this, on the console you will see

```
root : INFO Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO How quickly daft jumping zebras vex.
myapp.area2 : WARNING Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR The five boxing wizards jump quickly.
```

and in the file you will see something like

```
10-22 22:19 root INFO Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1 DEBUG Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1 INFO How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2 WARNING Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2 ERROR The five boxing wizards jump quickly.
```

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.
Configuration server example

Here is an example of a module using the logging configuration server:

```python
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999

try:
    t = logging.config.listen(9999)
    t.start()

    logger = logging.getLogger('simpleExample')

    try:
        # loop through logging calls to see the different
        # new configurations make, until Ctrl+C is pressed
        while True:
            logger.debug('debug message')
            logger.info('info message')
            logger.warning('warn message')
            logger.error('error message')
            logger.critical('critical message')
            time.sleep(5)
    except KeyboardInterrupt:
        # cleanup
        logging.config.stopListening()
        t.join()
```

And here is a script that takes a filename and sends that file to the
server, properly preceded with the binary-encoded length, as the new logging configuration:

```python
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```
Dealing with handlers that block

Sometimes you have to get your logging handlers to do their work without blocking the thread you're logging from. This is common in Web applications, though of course it also occurs in other scenarios.

A common culprit which demonstrates sluggish behaviour is the `SMTPHandler`: sending emails can take a long time, for a number of reasons outside the developer's control (for example, a poorly performing mail or network infrastructure). But almost any network-based handler can block: Even a `SocketHandler` operation may do a DNS query under the hood which is too slow (and this query can be deep in the socket library code, below the Python layer, and outside your control).

One solution is to use a two-part approach. For the first part, attach only a `QueueHandler` to those loggers which are accessed from performance-critical threads. They simply write to their queue, which can be sized to a large enough capacity or initialized with no upper bound to their size. The write to the queue will typically be accepted quickly, though you will probably need to catch the `queue.Full` exception as a precaution in your code. If you are a library developer who has performance-critical threads in their code, be sure to document this (together with a suggestion to attach only `QueueHandlers` to your loggers) for the benefit of other developers who will use your code.

The second part of the solution is `QueueListener`, which has been designed as the counterpart to `QueueHandler`. A `QueueListener` is very simple: it's passed a queue and some handlers, and it fires up an internal thread which listens to its queue for LogRecords sent from
QueueHandlers (or any other source of LogRecords, for that matter). The LogRecords are removed from the queue and passed to the handlers for processing.

The advantage of having a separate QueueListener class is that you can use the same instance to service multiple QueueHandlers. This is more resource-friendly than, say, having threaded versions of the existing handler classes, which would eat up one thread per handler for no particular benefit.

An example of using these two classes follows (imports omitted):

```python
que = queue.Queue(-1)  # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated the event (the main thread) rather than the inter
# thread which monitors the internal queue. This is what you want to happen.
root.warning('Look out!')
listener.stop()
```

which, when run, will produce:

```
MainThread: Look out!
```

Changed in version 3.5: Prior to Python 3.5, the QueueListener always passed every message received from the queue to every
handler it was initialized with. (This was because it was assumed that level filtering was all done on the other side, where the queue is filled.) From 3.5 onwards, this behaviour can be changed by passing a keyword argument `respect_handler_level=True` to the listener’s constructor. When this is done, the listener compares the level of each message with the handler’s level, and only passes a message to a handler if it’s appropriate to do so.
Sending and receiving logging events across a network

Let's say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a `SocketHandler` instance to the root logger at the sending end:

```python
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                                                  logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger:
logging.info('Jackdaws love my big sphinx of quartz.

# Now, define a couple of other loggers which might
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.

At the receiving end, you can set up a receiver using the
socketserver module. Here is a basic working example:

```python
import pickle
import logging
import logging.handlers
import socketserver
import struct

class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request."
    """
    This basically logs the record using whatever logging policy is configured locally.
    """
    def handle(self):
        """Handle multiple requests - each expected to be 4-byte length, followed by the LogRecord in pickle format. Logs the record according to whatever policy is configured locally."""
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)
```
def handleLogRecord(self, record):
    # if a name is specified, we use the named logger
    # implied by the record.
    if self.server.logname is not None:
        name = self.server.logname
    else:
        name = record.name
    logger = logging.getLogger(name)
    # N.B. EVERY record gets logged. This is because
    # is normally called AFTER logger-level filtering.
    # to do filtering, do it at the client end to
    # cycles and network bandwidth!
    logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    ""
    Simple TCP socket-based logging receiver suitable
    ""
    allow_reuse_address = True

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket], [], [],
                                        self.timeout)

            if rd:
```python
def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

First run the server, and then the client. On the client side, nothing is printed on the console; on the server side, you should see something like:

```
About to start TCP server...
  59 root    INFO    Jackdaws love my big
t  59 myapp.area1 DEBUG Quick zephyrs blow, v
  69 myapp.area1 INFO  How quickly daft jump
  69 myapp.area1 WARNING Jail zesty vixen who
  69 myapp.area2 ERROR  The five boxing wizard
```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the makePickle() method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.
Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client’s username, or IP address). Although you could use the extra parameter to achieve this, it’s not always convenient to pass the information in this way. While it might be tempting to create Logger instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of Logger instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of Logger instances becomes effectively unbounded.

Using LoggerAdapters to impart contextual information

An easy way in which you can pass contextual information to be output along with logging event information is to use the LoggerAdapter class. This class is designed to look like a Logger, so that you can call debug(), info(), warning(), error(), exception(), critical() and log(). These methods have the same signatures as their counterparts in Logger, so you can use the two types of instances interchangeably.

When you create an instance of LoggerAdapter, you pass it a Logger instance and a dict-like object which contains your contextual information. When you call one of the logging methods on an instance...
of LoggerAdapter, it delegates the call to the underlying instance of Logger passed to its constructor, and arranges to pass the contextual information in the delegated call. Here's a snippet from the code of LoggerAdapter:

```python
def debug(self, msg, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, contextual information from this adapter instance
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

The process() method of LoggerAdapter is where the contextual information is added to the logging output. It's passed the message and keyword arguments of the logging call, and it passes back (potentially) modified versions of these to use in the call to the underlying logger. The default implementation of this method leaves the message alone, but inserts an ‘extra’ key in the keyword argument whose value is the dict-like object passed to the constructor. Of course, if you had passed an ‘extra’ keyword argument in the call to the adapter, it will be silently overwritten.

The advantage of using ‘extra’ is that the values in the dict-like object are merged into the LogRecord instance's __dict__, allowing you to use customized strings with your Formatter instances which know about the keys of the dict-like object. If you need a different method, e.g. if you want to prepend or append the contextual information to the message string, you just need to subclass LoggerAdapter and override process() to do what you need. Here is a simple example:

```python
class CustomAdapter(logging.LoggerAdapter):
```
This example adapter expects the passed in dict-like object to have a 'connid' key, whose value in brackets is prepended to the log message.

```python
def process(self, msg, kwargs):
    return '[%s] %s' % (self.extra['connid'], msg)
```

which you can use like this:

```python
logger = logging.getLogger(__name__)
adaptor = CustomAdapter(logger, {'connid': some_conn_id})
```

Then any events that you log to the adapter will have the value of `some_conn_id` prepended to the log messages.

Using objects other than dicts to pass contextual information

You don’t need to pass an actual dict to a LoggerAdapter—you could pass an instance of a class which implements `__getitem__` and `__iter__` so that it looks like a dict to logging. This would be useful if you want to generate values dynamically (whereas the values in a dict would be constant).

Using Filters to impart contextual information

You can also add contextual information to log output using a user-defined Filter. Filter instances are allowed to modify the LogRecords passed to them, including adding additional attributes which can then be output using a suitable format string, or if needed a custom Formatter.

For example in a web application, the request being processed (or at
least, the interesting parts of it) can be stored in a threadlocal (`threading.local`) variable, and then accessed from a `Filter` to add, say, information from the request - say, the remote IP address and remote user's username - to the `LogRecord`, using the attribute names ‘ip’ and ‘user’ as in the `LoggerAdapter` example above. In that case, the same format string can be used to get similar output to that shown above. Here’s an example script:

```python
import logging
from random import choice

class ContextFilter(logging.Filter):
    ""
    This is a filter which injects contextual information
    Rather than use actual contextual information, we just use random data in this demo.
    ""

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.

    def filter(self, record):
        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %%(name)-5s %%(levelname)-8s
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
```
```python
a1.addFilter(f)
a2.addFilter(f)
a1.debug('A debug message')
a1.info('An info message with %s', 'some parameter')
for x in range(10):
    lvl = choice(levels)
    lvlname = logging.getLoggerName(lvl)
a2.log(lvl, 'A message at %s level with %d %s')
```

which, when run, produces something like:

```
2010-09-06 22:38:15,292 a.b.c DEBUG   IP: 123.231.231.123
2010-09-06 22:38:15,300 a.b.c INFO    IP: 192.168.0.1
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1
2010-09-06 22:38:15,300 d.e.f ERROR   IP: 127.0.0.1
2010-09-06 22:38:15,300 d.e.f DEBUG   IP: 127.0.0.1
2010-09-06 22:38:15,300 d.e.f ERROR   IP: 123.231.231.123
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1
2010-09-06 22:38:15,300 d.e.f DEBUG   IP: 192.168.0.1
2010-09-06 22:38:15,300 d.e.f ERROR   IP: 127.0.0.1
2010-09-06 22:38:15,300 d.e.f DEBUG   IP: 123.231.231.123
2010-09-06 22:38:15,300 d.e.f INFO    IP: 123.231.231.231
```
Logging to a single file from multiple processes

Although logging is thread-safe, and logging to a single file from multiple threads in a single process is supported, logging to a single file from multiple processes is not supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple processes, one way of doing this is to have all the processes log to a SocketHandler, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) This section documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

If you are using a recent version of Python which includes the multiprocessing module, you could write your own handler which uses the Lock class from this module to serialize access to the file from your processes. The existing FileHandler and subclasses do not make use of multiprocessing at present, though they may do so in the future. Note that at present, the multiprocessing module does not provide working lock functionality on all platforms (see https://bugs.python.org/issue3770).

Alternatively, you can use a Queue and a QueueHandler to send all logging events to one of the processes in your multi-process application. The following example script demonstrates how you can do this; in the example a separate listener process listens for events sent by other processes and logs them according to its own logging configuration. Although the example only demonstrates one way of
doing it (for example, you may want to use a listener thread rather than a separate listener process – the implementation would be analogous) it does allow for completely different logging configurations for the listener and the other processes in your application, and can be used as the basis for code meeting your own specific requirements:

```python
# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

# Because you'll want to define the logging configurations for
# listener and worker process functions take a configurer parameter
# which they use for communication.

# In practice, you can configure the listener however you want,
# In practice, you would probably want to do this logic in the
# The size of the rotated files is made small so you can see the
# This is the listener process top-level loop: wait
# (LogRecords) on the queue and handle them, quit when
```
# LogRecord.

def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # We send this as a sentinel to tell the listener to quit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # No level or filter logic applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

    # Arrays used for random selections in this demo

    LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL]

    LOGGERS = ['a.b.c', 'd.e.f']

    MESSAGES = [
        'Random message #1',
        'Random message #2',
        'Random message #3',
    ]

    # The worker configuration is done at the start of the worker process run.
    # Note that on Windows you can't rely on fork semantics, so each process
    # will run the logging configuration code when it starts.

def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Just one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
    root.setLevel(logging.DEBUG)
This is the worker process top-level loop, which just logs ten events with random intervening delays before terminating. The print messages are just so you know it's doing something!

```python
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
        print('Worker finished: %s' % name)

Here's where the demo gets orchestrated. Create the listener, create ten workers and start them, wait for them to finish, then send a None to the queue to tell the listener.

```python
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process, args=(queue,))
    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process, args=(queue,))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```
A variant of the above script keeps the logging in the main process, in a separate thread:

```python
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)

def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz', 'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
```

q = Queue()
d = {
    'version': 1,
    'formatters': {
        'detailed': {
            'class': 'logging.Formatter',
            'format': '%(asctime)s %(name)-15s %(levelname)-8s
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
        },
        'file': {
            'class': 'logging.FileHandler',
            'filename': 'mplog.log',
            'mode': 'w',
            'formatter': 'detailed',
        },
        'foofile': {
            'class': 'logging.FileHandler',
            'filename': 'mplog-foo.log',
            'mode': 'w',
            'formatter': 'detailed',
        },
        'errors': {
            'class': 'logging.FileHandler',
            'filename': 'mplog-errors.log',
            'mode': 'w',
            'level': 'ERROR',
            'formatter': 'detailed',
        },
    },
    'loggers': {
        'foo': {
            'handlers': ['foofile']
        },
    },
}
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker workers'.append(wp)
    wp.start()
logging.config.dictConfig(d)
lp = threading.Thread(target=logger_thread, args=
lp.start()
# At this point, the main process could do some
# Once it's done that, it can wait for the workers
def worker_process():
    q.put('worker done')
    wp.join()
# And now tell the logging thread to finish up,
q.put(None)
lp.join()

This variant shows how you can e.g. apply configuration for particular loggers - e.g. the foo logger has a special handler which stores all events in the foo subsystem in a file mplog-foo.log. This will be used by the logging machinery in the main process (even though the logging events are generated in the worker processes) to direct the messages to the appropriate destinations.
Using file rotation

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a RotatingFileHandler:

```python
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME, maxBytes=20, backupCount=

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```
The result should be 6 separate files, each with part of the log history for the application:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (`.1` becomes `.2`, etc.) and the `.6` file is erased.

Obviously this example sets the log length much too small as an extreme example. You would want to set `maxBytes` to an appropriate value.
Use of alternative formatting styles

When logging was added to the Python standard library, the only way of formatting messages with variable content was to use the %-formatting method. Since then, Python has gained two new formatting approaches: `string.Template` (added in Python 2.4) and `str.format()` (added in Python 2.6).

Logging (as of 3.2) provides improved support for these two additional formatting styles. The Formatter class been enhanced to take an additional, optional keyword parameter named `style`. This defaults to '%', but other possible values are '{' and '$', which correspond to the other two formatting styles. Backwards compatibility is maintained by default (as you would expect), but by explicitly specifying a style parameter, you get the ability to specify format strings which work with `str.format()` or `string.Template`. Here's an example console session to show the possibilities:

```python
>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}.
...                        style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL message
>>> df = logging.Formatter('${asctime} $name ${levelname} ${message}.
...                        style='$')
```
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message

Note that the formatting of logging messages for final output to logs is completely independent of how an individual logging message is constructed. That can still use %-formatting, as shown here:

>>> logger.error('This is an%ss %s %s', 'other,', 'ERROR,')
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR,

Logging calls (logger.debug(), logger.info() etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the actual logging call (e.g. the exc_info keyword parameter to indicate that traceback information should be logged, or the extra keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using str.format() or string.Template syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There is, however, a way that you can use {}- and $- formatting to construct your individual log messages. Recall that for a message you can use an arbitrary object as a message format string, and that the logging package will call str() on that object to get the actual format
string. Consider the following two classes:

```python
class BraceMessage:
    def __init__(self, fmt, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or $-
formatting to be used to build the actual “message” part which appears
in the formatted log output in place of “%(message)s” or “{message}”
or “$message”. It’s a little unwieldy to use the class names whenever
you want to log something, but it’s quite palatable if you use an alias
such as __ (double underscore — not to be confused with _, the single
underscore used as a synonym/alias for gettext.gettext() or its
brethren).

The above classes are not included in Python, though they’re easy
enough to copy and paste into your own code. They can be used as
follows (assuming that they’re declared in a module called
wherever):
>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2, name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders

While the above examples use print() to show how the formatting works, you would of course use logger.debug() or similar to actually log using this approach.

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the __ notation is just syntax sugar for a constructor call to one of the XXXMessage classes.

If you prefer, you can use a LoggerAdapter to achieve a similar effect to the above, as in the following example:

```python
import logging
```
```python
class Message(object):
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)

class StyleAdapter(logging.LoggerAdapter):
    def __init__(self, logger, extra=None):
        super(StyleAdapter, self).__init__(logger, extra)

    def log(self, level, msg, *args, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger._log(level, Message(msg, args)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
    logger.debug('Hello, {}', 'world!')

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    main()
```

The above script should log the message **Hello, world!** when run with Python 3.2 or later.
Customizing LogRecord

Every logging event is represented by a `LogRecord` instance. When an event is logged and not filtered out by a logger's level, a `LogRecord` is created, populated with information about the event and then passed to the handlers for that logger (and its ancestors, up to and including the logger where further propagation up the hierarchy is disabled). Before Python 3.2, there were only two places where this creation was done:

- `Logger.makeRecord()`, which is called in the normal process of logging an event. This invoked `LogRecord` directly to create an instance.
- `makeLogRecord()`, which is called with a dictionary containing attributes to be added to the LogRecord. This is typically invoked when a suitable dictionary has been received over the network (e.g. in pickle form via a `SocketHandler`, or in JSON form via an `HTTPHandler`).

This has usually meant that if you need to do anything special with a `LogRecord`, you've had to do one of the following.

- Create your own `Logger` subclass, which overrides `Logger.makeRecord()`, and set it using `setLoggerClass()` before any loggers that you care about are instantiated.
- Add a `Filter` to a logger or handler, which does the necessary special manipulation you need when its `filter()` method is called.

The first approach would be a little unwieldy in the scenario where (say) several different libraries wanted to do different things. Each would attempt to set its own `Logger` subclass, and the one which did
this last would win.

The second approach works reasonably well for many cases, but does not allow you to e.g. use a specialized subclass of LogRecord. Library developers can set a suitable filter on their loggers, but they would have to remember to do this every time they introduced a new logger (which they would do simply by adding new packages or modules and doing

```python
def logger = logging.getLogger(__name__)
```

at module level). It's probably one too many things to think about. Developers could also add the filter to a NullHandler attached to their top-level logger, but this would not be invoked if an application developer attached a handler to a lower-level library logger — so output from that handler would not reflect the intentions of the library developer.

In Python 3.2 and later, LogRecord creation is done through a factory, which you can specify. The factory is just a callable you can set with setLogRecordFactory(), and interrogate with getLogRecordFactory(). The factory is invoked with the same signature as the LogRecord constructor, as LogRecord is the default setting for the factory.

This approach allows a custom factory to control all aspects of LogRecord creation. For example, you could return a subclass, or just add some additional attributes to the record once created, using a pattern similar to this:

```python
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
```
record = old_factory(*args, **kwargs)
record.custom_attribute = 0xdecafbad
return record

logging.setLogRecordFactory(record_factory)

This pattern allows different libraries to chain factories together, and as long as they don't overwrite each other's attributes or unintentionally overwrite the attributes provided as standard, there should be no surprises. However, it should be borne in mind that each link in the chain adds run-time overhead to all logging operations, and the technique should only be used when the use of a Filter does not provide the desired result.
Subclassing QueueHandler - a ZeroMQ example

You can use a QueueHandler subclass to send messages to other kinds of queues, for example a ZeroMQ 'publish' socket. In the example below, the socket is created separately and passed to the handler (as its 'queue'):

```python
import zmq  # using pyzmq, the Python binding for ZeroMQ
import json  # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB)  # or zmq.PUSH, or other
sock.bind('tcp://*:5556')  # or wherever

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

handler = ZeroMQSocketHandler(sock)
```

Of course there are other ways of organizing this, for example passing in the data needed by the handler to create the socket:

```python
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
        socket.bind(uri)
        super().__init__(socket)

    def enqueue(self, record):
```
self.queue.send_json(record.__dict__)

def close(self):
    self.queue.close()
Subclassing QueueListener - a ZeroMQ example

You can also subclass QueueListener to get messages from other kinds of queues, for example a ZeroMQ 'subscribe' socket. Here's an example:

```python
class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt_string(zmq.SUBSCRIBE, '')
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self):
        msg = self.queue.recv_json()
        return logging.makeLogRecord(msg)
```

See also:

**Module logging**

API reference for the logging module.

**Module logging.config**

Configuration API for the logging module.

**Module logging.handlers**

Useful handlers included with the logging module.

A basic logging tutorial

A more advanced logging tutorial
An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it’s taken from the documentation on the Django project. This dictionary is passed to `dictConfig()` to put the configuration into effect:

```python
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'django.utils.log.NullHandler',
        },
        'console':{
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
        }
    },
}
```
'class': 'django.utils.log.AdminEmailHandler',
'filters': ['special']
}
},
'loggers': {
'django': {
'handlers': ['null'],
'propagate': True,
'level': 'INFO',
},
django.request': {
'handlers': ['mail_admins'],
'level': 'ERROR',
'propagate': False,
},
'myproject.custom': {
'handlers': ['console', 'mail_admins'],
'level': 'INFO',
'filters': ['special']
}
}
}

For more information about this configuration, you can see the relevant section of the Django documentation.
Using a rotator and namer to customize log rotation processing

An example of how you can define a namer and rotator is given in the following snippet, which shows zlib-based compression of the log file:

```python
def namer(name):
    return name + " .gz"

def rotator(source, dest):
    with open(source, "rb") as sf:
        data = sf.read()
        compressed = zlib.compress(data, 9)
    with open(dest, "wb") as df:
        df.write(compressed)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler(...)
rh.rotator = rotator
rh.namer = namer
```

These are not “true” .gz files, as they are bare compressed data, with no “container” such as you’d find in an actual gzip file. This snippet is just for illustration purposes.
A more elaborate multiprocessing example

The following working example shows how logging can be used with multiprocessing using configuration files. The configurations are fairly simple, but serve to illustrate how more complex ones could be implemented in a real multiprocessing scenario.

In the example, the main process spawns a listener process and some worker processes. Each of the main process, the listener and the workers have three separate configurations (the workers all share the same configuration). We can see logging in the main process, how the workers log to a QueueHandler and how the listener implements a QueueListener and a more complex logging configuration, and arrange to dispatch events received via the queue to the handlers specified in the configuration. Note that these configurations are purely illustrative, but you should be able to adapt this example to your own scenario.

Here’s the script - the docstrings and the comments hopefully explain how it works:

```python
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue, Event, current_process
import os
import random
import time

class MyHandler:
    """
    A simple handler for logging events. It runs in
    dispatches events to loggers based on the name i
```
which then get dispatched, by the logging system configured for those loggers.

```python
def handle(self, record):
    logger = logging.getLogger(record.name)
    # The process name is transformed just to show that it's the listener doing the logging to files and console
    record.processName = '%s (for %s)' % (current_process, logger.handle(record))

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in a separate process for illustrative purposes.

    This initialises logging according to the specified configuration, starts the listener and waits for the main process to signal completion via the event. The listener is then stopped, and the process exits.
    """
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler)
    listener.start()
    if os.name == 'posix':
        # On POSIX, the setup logger will have been parent process, but should have been disabled by dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't exist in the child, so it would be created and would appear - hence the "if posix" clause
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of
practice, they could be a heterogeneous bunch of processes which are identical to each other.

This initialises logging according to the specified configuration, and logs a hundred messages with random levels to randomly selected loggers.

A small sleep is added to allow other processes a chance to run. This is not strictly needed, but it mixes the output from the different processes a bit more than if it's left out.

```
logging.config.dictConfig(config)
levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.CRITICAL]
loggers = ['foo', 'foo.bar', 'foo.bar.baz', 'spam', 'spam.ham', 'spam.ham.eggs']
if os.name == 'posix':
    # On POSIX, the setup logger will have been
    # parent process, but should have been disabled
    # dictConfig call.
    # On Windows, since fork isn't used, the setup logger won't
    # exist in the child, so it would be created
    # would appear - hence the "if posix" clause
    logger = logging.getLogger('setup')
    logger.critical('Should not appear, because

    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
        time.sleep(0.01)
```

def main():
    q = Queue()
    # The main process gets a simple configuration
    config_initial = {
        'version': 1,
        'formatters': {
'detailed': {
    'class': 'logging.Formatter',
    'format': '%(asctime)s %(name)-15s %(levelname)-8s

},

'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'level': 'INFO',
    },
},

'root': {
    'level': 'DEBUG',
    'handlers': ['console']
}
}

# The worker process configuration is just a Queue root logger, which allows all messages to be sent to a queue.
# We disable existing loggers to disable the "setup" logger used in the parent process. This is needed on POSIX because
# be there in the child following a fork().
config_worker = {
    'version': 1,
    'disable_existing_loggers': True,
    'handlers': {
        'queue': {
            'class': 'logging.handlers.QueueHandler',
            'queue': q,
        },
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['queue']
    }
}

# The listener process configuration shows that the full flexibility of logging configuration is available to dispatch
# you want.
# We disable existing loggers to disable the "setup" logger used in the
# parent process. This is needed on POSIX because the logger will
# be there in the child following a fork().

```python
cfg = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'detailed': {
            'class': 'logging.Formatter',
            'format': '%(asctime)s %(name)-15s %(levelname)-8s %%(message)s
        },
        'simple': {
            'class': 'logging.Formatter',
            'format': '%(name)-15s %(levelname)-8s %%(message)s
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'simple'
        },
        'file': {
            'class': 'logging.FileHandler',
            'filename': 'mplog.log',
            'mode': 'w',
            'formatter': 'detailed'
        },
        'foofile': {
            'class': 'logging.FileHandler',
            'filename': 'mplog-foo.log',
            'mode': 'w',
            'formatter': 'detailed'
        },
        'errors': {
            'class': 'logging.FileHandler',
```
'filename': 'mplog-errors.log',
'mode': 'w',
'level': 'ERROR',
'formatter': 'detailed',
},
},
'loggers': {
'foo': {
'handlers': ['foofile']
}
},
'root': {
'level': 'DEBUG',
'handlers': ['console', 'file', 'errors'
}
},
}

# Log some initial events, just to show that logging in the parent works normally.
logging.config.dictConfig(config_initial)
logger = logging.getLogger('setup')
logger.info('About to create workers...')
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker args=(config_worker,))
    workers.append(wp)
    wp.start()
    logger.info('Started worker: %s', wp.name)
logger.info('About to create listener...')
stop_event = Event()
lp = Process(target=listener_process, name='listener args=(q, stop_event, config_listener
lp.start()
logger.info('Started listener')
# We now hang around for the workers to finish their work.
for wp in workers:
    wp.join()
# Workers all done, listening can now stop.
# Logging in the parent still works normally.
logger.info('Telling listener to stop ...')
stop_event.set()
lp.join()
logger.info('All done.')

if __name__  == '__main__':
    main()
Inserting a BOM into messages sent to a SysLogHandler

**RFC 5424** requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the relevant section of the specification.)

In Python 3.1, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 3.2.4 and later. However, it is not being replaced, and if you want to produce **RFC 5424**-compliant messages which include a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a `Formatter` instance to your `SysLogHandler` instance, with a format string such as:

   ```python
   'ASCII section\ufeffUnicode section'
   ```

   The Unicode code point U+FEFF, when encoded using UTF-8, will be encoded as a UTF-8 BOM – the byte-string `b'\xef\xbb\xbf'`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after substitution
is always ASCII (that way, it will remain unchanged after UTF-8 encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that’s fine – it will be encoded using UTF-8.

The formatted message will be encoded using UTF-8 encoding by SysLogHandler. If you follow the above rules, you should be able to produce RFC 5424-compliant messages. If you don’t, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.
Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which is capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```python
import json
import logging

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage  # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num='123'))
```

If the above script is run, it prints:

```plaintext
message 1 >>> {'fnum': 123.456, 'num': 123, 'bar': 'baz'}
```
Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom JSON encoder, as in the following complete example:

```python
from __future__ import unicode_literals
import json
import logging

# This next bit is to ensure the script runs unchanged
try:
    unicode
except NameError:
    unicode = str

class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            return o.encode('unicode_escape').decode
        return super(Encoder, self).default(o)

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwars = kwars

    def __str__(self):
        s = Encoder().encode(self.kwars)
        return '%s >>> %s' % (self.message, s)

_= StructuredMessage  # optional, to improve readability
```
```python
def main():
    logging.basicConfig(level=logging.INFO, format='logging.info(_("message 1", set_value={1, 2, 3}),
    if __name__ == '__main__':
        main()
```

When the above script is run, it prints:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1,
```

Note that the order of items might be different according to the version of Python used.
Customizing handlers with `dictConfig()`

There are times when you want to customize logging handlers in particular ways, and if you use `dictConfig()` you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using `shutil.chown()`, but the file handlers in the stdlib don’t offer built-in support. You can customize handler creation using a plain function such as:

```python
def owned_file_handler(filename, mode='a', encoding=None, owner=True):
    if not os.path.exists(filename):
        open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

You can then specify, in a logging configuration passed to `dictConfig()`, that a logging handler be created by calling this function:

```python
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s
        },
    },
    'handlers': {
        'file': {
```
In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```python
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding='utf-8', owner='pulse', group='pulse'):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, owner=owner)
    return logging.FileHandler(filename, mode, encoding=encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    }
```

```python
'formatters': {
    'default': {
        'format': '%(asctime)s %(levelname)s %(name)s
    },
},
'handlers': {
    'file': {
        # The values below are popped from this
dict and used to create the handler, set the handler's level
        # and its formatter.
        '()': owned_file_handler,
        'level': 'DEBUG',
        'formatter': 'default',
        # The values below are passed to the handler creator callable
        # as keyword arguments.
        'owner': ['pulse', 'pulse'],
        'filename': 'chowntest.log',
        'mode': 'w',
        'encoding': 'utf-8',
    },
},
'root': {
    'handlers': ['file'],
    'level': 'DEBUG',
}
}
logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')
```

To run this, you will probably need to run as root:

```
$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
```
Note that this example uses Python 3.3 because that's where `shutil.chown()` makes an appearance. This approach should work with any Python version that supports `dictConfig()` - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. `os.chown()`.

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

```python
'()': owned_file_handler,
```

you could use e.g.:

```python
'()': 'ext://project.util.owned_file_handler',
```

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `ext://__main__.owned_file_handler` should work. Here, the actual callable is resolved by `dictConfig()` from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using `os.chmod()`.

Of course, the approach could also be extended to types of handler other than a `FileHandler` - for example, one of the rotating file handlers, or a different type of handler altogether.
Using particular formatting styles throughout your application

In Python 3.2, the `Formatter` gained a `style` keyword parameter which, while defaulting to `%` for backward compatibility, allowed the specification of `{` or `$` to support the formatting approaches supported by `str.format()` and `string.Template`. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

Logging calls (`debug()`, `info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using `str.format()` or `string.Template` syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using %-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of
the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

**Using LogRecord factories**

In Python 3.2, along with the *Formatter* changes mentioned above, the logging package gained the ability to allow users to set their own *LogRecord* subclasses, using the *setLogRecordFactory()* function. You can use this to set your own subclass of *LogRecord*, which does the Right Thing by overriding the *getMessage()* method. The base class implementation of this method is where the `msg % args` formatting happens, and where you can substitute your alternate formatting; however, you should be careful to support all formatting styles and allow `%`-formatting as the default, to ensure interoperability with other code. Care should also be taken to call `str(self.msg)`, just as the base implementation does.

Refer to the reference documentation on *setLogRecordFactory()* and *LogRecord* for more information.

**Using custom message objects**

There is another, perhaps simpler way that you can use {}- and $-formatting to construct your individual log messages. You may recall (from *Using arbitrary objects as messages*) that when logging you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```python
class BraceMessage(object):
    def __init__(self, fmt, *args, **kwargs):
```
Either of these can be used in place of a format string, to allow { }- or $-formatting to be used to build the actual “message” part which appears in the formatted log output in place of “%(message)s” or “{message}” or “$message”. If you find it a little unwieldy to use the class names whenever you want to log something, you can make it more palatable if you use an alias such as M or _ for the message (or perhaps __, if you are using _ for localization).

Examples of this approach are given below. Firstly, formatting with str.format():

```python
>>> __ = BraceMessage
>>> print(__('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
```
Secondly, formatting with `string.Template`:

```python
>>> __ = DollarMessage
>>> print(__('Message with $num $what', num=2, what=
Message with 2 placeholders
>>> 
```

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the `__` notation is just syntax sugar for a constructor call to one of the `XXXMessage` classes shown above.
Configuring filters with `dictConfig()`

You can configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it's only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```python
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

LOGGING = {
    'version': 1,
    'filters': {
```
{'myfilter': {
    '(': MyFilter,
    'param': 'noshow',
},
},
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'filters': ['myfilter']
    }
},
'root': {
    'level': 'DEBUG',
    'handlers': ['console']
},
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

changed: hello

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form ext://... as described in Access to external objects. For example, you
could have used the text `ext://__main__.MyFilter` instead of MyFilter in the above example.

- As well as for filters, this technique can also be used to configure custom handlers and formatters. See User-defined objects for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe Customizing handlers with dictConfig() above.
Customized exception formatting

There might be times when you want to do customized exception formatting - for argument's sake, let's say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```python
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super(OneLineExceptionFormatter, self).formatException(exc_info)
        return repr(result)  # or format into one line however you want to

def format(self, record):
    s = super(OneLineExceptionFormatter, self).format(record)
    if record.exc_text:
        s = s.replace('\n', '') + '|
    return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(name)s|%(funcName)s|%(message)s', '
    %d/%m/%Y %H:%M:%S

    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
```
```python
logging.info('Sample message')
try:
    x = 1 / 0
except ZeroDivisionError as e:
    logging.exception('ZeroDivisionError: %s', e)
if __name__ == '__main__':
    main()
```

When run, this produces a file with exactly two lines:

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: integer division or modulo by zero|
```

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.
Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system, even if it doesn’t have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using `subprocess`. It’s assumed here that TTS command line programs won’t expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it’s acceptable to have the messages spoken one at a time rather than concurrently. The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available:

```python
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.STDOUT,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

    def configure_logging():
```

```sql
```
h = TTSHandler()
root = logging.getLogger()
root.addHandler(h)
# the default formatter just returns the message
root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
sys.exit(main())

When run, this script should say “Hello” and then “Goodbye” in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.
Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don’t want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the `logging.handlers.MemoryHandler`, which allows buffering of logged events until some condition occurs, at which point the buffered events are flushed - passed to another handler (the target handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it's about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that's required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call...
to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer. These default to a StreamHandler which writes to `sys.stderr`, `logging.ERROR` and `100` respectively.

Here’s the script:

```python
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level)

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
            logger.removeHandler(handler)
        return wrapper
```

return decorator

def write_line(s):
    sys.stderr.write('%s
' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)

When this script is run, the following output should be observed:

Calling undecorated foo with False
As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration:

```python
@log_if_errors(logger)
def foo(fail=False):
    ...
```
Formatting times using UTC (GMT) via configuration

Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below:

```python
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

and you can then use the `UTCFormatter` in your code instead of `Formatter`. If you want to do that via configuration, you can use the `dictConfig()` API with an approach illustrated by the following complete example:

```python
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        },
    }
}
```

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())

When this script is run, it should print something like:

2015-10-17 12:53:29,501 The local time is Sat Oct 17
2015-10-17 13:53:29,501 The local time is Sat Oct 17

showing how the time is formatted both as local time and UTC, one for each handler.
Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager:

```python
import logging
import sys

class LoggingContext(object):
    def __init__(self, logger, level=None, handler=None):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
            if self.handler:
                self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
```
If you specify a level value, the logger's level is set to that value in the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don't need the handler any more.

To illustrate how it works, we can add the following block of code to the above:

```python
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h):
        logger.debug('5. This should appear twice - one on stderr and one on stdout.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger's level to `INFO`, so message #1 appears and message #2 doesn't. We then change the level to `DEBUG` temporarily in the following with block, and so message #3 appears. After the block exits, the logger's level is restored to `INFO` and so message #4 doesn't appear. In the next with block, we set the level to `DEBUG` again but also add a handler writing to `sys.stdout`. Thus, message
#5 appears twice on the console (once via stderr and once via stdout). After the with statement’s completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn’t (just like message #2).

If we run the resulting script, the result is as follows:

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

If we run it again, but pipe stderr to /dev/null, we see the following, which is the only message written to stdout:

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.
```

Once again, but piping stdout to /dev/null, we get:

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

In this case, the message #5 printed to stdout doesn’t appear, as expected.

Of course, the approach described here can be generalised, for
example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.
Regular Expression HOWTO

Author: A.M. Kuchling <amk@amk.ca>

Abstract

This document is an introductory tutorial to using regular expressions in Python with the `re` module. It provides a gentler introduction than the corresponding section in the Library Reference.
Introduction

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.
Simple Patterns

We'll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression `test` will match the string `test` exactly. (You can enable a case-insensitive mode that would let this RE match `Test` or `TEST` as well; more about this later.)

There are exceptions to this rule; some characters are special metacharacters, and don’t match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do.

Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

```
.  ^  $  *  +  ?  {  }  [  ]  \  |  (  )
```

The first metacharacters we'll look at are `[]` and `]`. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of
characters can be indicated by giving two characters and separating them by a '-'. For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z].

Metacharacters are not active inside classes. For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside a character class it's stripped of its special nature.

You can match the characters not listed within the class by complementing the set. This is indicated by including a '^' as the first character of the class. For example, [^5] will match any character except '5'. If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5^] will match either a '5' or a '^'.

Perhaps the most important metacharacter is the backslash, \. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \\ or \\\\.

Some of the special sequences beginning with '\' represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace.

Let's take an example: \w matches any alphanumeric character. If the regex pattern is expressed in bytes, this is equivalent to the class [a-zA-Z0-9_]. If the regex pattern is a string, \w will match all the
characters marked as letters in the Unicode database provided by the `unicodedata` module. You can use the more restricted definition of \w in a string pattern by supplying the `re.ASCII` flag when compiling the regular expression.

The following list of special sequences isn’t complete. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of *Regular Expression Syntax* in the Standard Library reference. In general, the Unicode versions match any character that’s in the appropriate category in the Unicode database.

\d
   Matches any decimal digit; this is equivalent to the class `[0-9]`.

\D
   Matches any non-digit character; this is equivalent to the class `[^0-9]`.

\s
   Matches any whitespace character; this is equivalent to the class `[^\t\n\r\f\v]`.

\S
   Matches any non-whitespace character; this is equivalent to the class `[^\t\n\r\f\v]`.

\w
   Matches any alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

\W
   Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For
example, \[s,.\] is a character class that will match any whitespace character, or ', ' or '. '

The final metacharacter in this section is . It matches anything except a newline character, and there’s an alternate mode (\texttt{re.DOTALL}) where it will match even a newline. . is often used where you want to match “any character”.

**Repeating Things**

Being able to match varying sets of characters is the first thing regular expressions can do that isn’t already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn’t be much of an advance. Another capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we’ll look at is *. * doesn’t match the literal character '*'; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, ca*t will match 'ct' (0 'a' characters), 'cat' (1 'a'), 'caaat' (3 'a' characters), and so forth.

Repetitions such as * are greedy; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don’t match, the matching engine will then back up and try again with fewer repetitions.

A step-by-step example will make this more obvious. Let’s consider the expression \texttt{a[bc\textbackslash d]*b}. This matches the letter \texttt{'a'}, zero or more
letters from the class \([bcd]\), and finally ends with a 'b'. Now imagine matching this RE against the string 'abcd'.

<table>
<thead>
<tr>
<th>Step</th>
<th>Matched</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>The a in the RE matches.</td>
</tr>
<tr>
<td>2</td>
<td>abcbd</td>
<td>The engine matches ([bcd]^{*}), going as far as it can, which is to the end of the string.</td>
</tr>
<tr>
<td>3</td>
<td>Failure</td>
<td>The engine tries to match b, but the current position is at the end of the string, so it fails.</td>
</tr>
<tr>
<td>4</td>
<td>abcb</td>
<td>Back up, so that ([bcd]^{*}) matches one less character.</td>
</tr>
<tr>
<td>5</td>
<td>Failure</td>
<td>Try b again, but the current position is at the last character, which is a 'd'.</td>
</tr>
<tr>
<td>6</td>
<td>abc</td>
<td>Back up again, so that ([bcd]^{*}) is only matching bc.</td>
</tr>
<tr>
<td>6</td>
<td>abcb</td>
<td>Try b again. This time the character at the current position is 'b', so it succeeds.</td>
</tr>
</tbody>
</table>

The end of the RE has now been reached, and it has matched 'abcd'. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for \([bcd]^{*}\), and if that subsequently fails, the engine will conclude that the string doesn’t match the RE at all.

Another repeating metacharacter is +, which matches one or more times. Pay careful attention to the difference between * and +; * matches zero or more times, so whatever’s being repeated may not be present at all, while + requires at least one occurrence. To use a similar example, ca+t will match 'cat' (1 'a'), 'caaat' (3 'a's),
but won’t match 'ct'.

There are two more repeating qualifiers. The question mark character, ?, matches either once or zero times; you can think of it as marking something as being optional. For example, home-?brew matches either 'homebrew' or 'home-brew'.

The most complicated repeated qualifier is \{m,n\}, where m and n are decimal integers. This qualifier means there must be at least m repetitions, and at most n. For example, a/\{1,3\}b will match 'a/b', 'a//b', and 'a///b'. It won’t match 'ab', which has no slashes, or 'a/////b', which has four.

You can omit either m or n; in that case, a reasonable value is assumed for the missing value. Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.

Readers of a reductionist bent may notice that the three other qualifiers can all be expressed using this notation. \{0,\} is the same as *, \{1,\} is equivalent to +, and \{0,1\} is the same as ?. It’s better to use *, +, or ? when you can, simply because they’re shorter and easier to read.
Using Regular Expressions

Now that we’ve looked at some simple regular expressions, how do we actually use them in Python? The re module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

Compiling Regular Expressions

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```python
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

`re.compile()` also accepts an optional `flags` argument, used to enable various special features and syntax variations. We’ll go over the available settings later, but for now a single example will do:

```python
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren’t part of the core Python language, and no special syntax was created for expressing them. (There are applications that don’t need REs at all, so there’s no need to bloat the language specification by including them.) Instead, the re module is simply a C extension module included with Python, just like the socket or zlib modules.
Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.

The Backslash Plague

As stated earlier, regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflicts with Python’s usage of the same character for the same purpose in string literals.

Let’s say you want to write a RE that matches the string \section, which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string \\section. The resulting string that must be passed to re.compile() must be \section. However, to express this as a Python string literal, both backslashes must be escaped again.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>\section</td>
<td>Text string to be matched</td>
</tr>
<tr>
<td>\section</td>
<td>Escaped backslash for re.compile()</td>
</tr>
<tr>
<td>&quot;\\section&quot;</td>
<td>Escaped backslashes for a string literal</td>
</tr>
</tbody>
</table>

In short, to match a literal backslash, one has to write '\\\' as the RE string, because the regular expression must be \, and each backslash must be expressed as \ inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes and makes the resulting strings difficult to
understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with 'r', so r"\n" is a two-character string containing '\' and 'n', while "\n" is a one-character string containing a newline. Regular expressions will often be written in Python code using this raw string notation.

In addition, special escape sequences that are valid in regular expressions, but not valid as Python string literals, now result in a DeprecationWarning and will eventually become a SyntaxError, which means the sequences will be invalid if raw string notation or escaping the backslashes isn't used.

<table>
<thead>
<tr>
<th>Regular String</th>
<th>Raw string</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ab*&quot;</td>
<td>r&quot;ab*&quot;</td>
</tr>
<tr>
<td>&quot;\section&quot;</td>
<td>r&quot;\section&quot;</td>
</tr>
<tr>
<td>&quot;\w+\s+\1&quot;</td>
<td>r&quot;\w+\s+\1&quot;</td>
</tr>
</tbody>
</table>

Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the re docs for a complete listing.

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>match()</td>
<td>Determine if the RE matches at the beginning of the string.</td>
</tr>
<tr>
<td>search()</td>
<td>Scan through a string, looking for any location</td>
</tr>
</tbody>
</table>
where this RE matches.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findall()</code></td>
<td>Find all substrings where the RE matches, and returns them as a list.</td>
</tr>
<tr>
<td><code>finditer()</code></td>
<td>Find all substrings where the RE matches, and returns them as an iterator.</td>
</tr>
</tbody>
</table>

`match()` and `search()` return `None` if no match can be found. If they're successful, a `match object` instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the `re` module. If you have `tkinter` available, you may also want to look at `Tools/demo/redemo.py`, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE.

This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the `re` module, and compile a RE:

```python
>>> import re
>>> p = re.compile('^[a-z]+')
>>> p
re.compile('^[a-z]+')
```

Now, you can try matching various strings against the RE `[a-z]+`. An empty string shouldn’t match at all, since `+` means ‘one or more repetitions’. `match()` should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of `match()` to make this clear.
Now, let's try it on a string that it should match, such as `tempo`. In this case, `match()` will return a match object, so you should store the result in a variable for later use.

```python
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Now you can query the match object for information about the matching string. Match object instances also have several methods and attributes; the most important ones are:

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>group()</td>
<td>Return the string matched by the RE</td>
</tr>
<tr>
<td>start()</td>
<td>Return the starting position of the match</td>
</tr>
<tr>
<td>end()</td>
<td>Return the ending position of the match</td>
</tr>
<tr>
<td>span()</td>
<td>Return a tuple containing the (start, end) positions of the match</td>
</tr>
</tbody>
</table>

Trying these methods will soon clarify their meaning:

```python
>>> m.group()
'tempo'
```

```python
>>> m.start(), m.end()
(0, 5)
```

```python
>>> m.span()
(0, 5)
```
group() returns the substring that was matched by the RE. start() and end() return the starting and ending index of the match. span() returns both start and end indexes in a single tuple. Since the match() method only checks if the RE matches at the start of a string, start() will always be zero. However, the search() method of patterns scans through the string, so the match may not start at zero in that case.

```python
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the match object in a variable, and then check if it was None. This usually looks like:

```python
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

Two pattern methods return all of the matches for a pattern. findAll() returns a list of matching strings:

```python
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping')
['12', '11', '10']
```
The r prefix, making the literal a raw string literal, is needed in this example because escape sequences in a normal “cooked” string literal that are not recognized by Python, as opposed to regular expressions, now result in a DeprecationWarning and will eventually become a SyntaxError. See The Backslash Plague.

findall() has to create the entire list before it can be returned as the result. The finditer() method returns a sequence of match object instances as an iterator:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ...
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...    print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Module-Level Functions

You don’t have to create a pattern object and call its methods; the re module also provides top-level functions called match(), search(), findall(), sub(), and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either None or a match object instance.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
```
Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE won’t need to parse the pattern again and again.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? If you’re accessing a regex within a loop, pre-compiling it will save a few function calls. Outside of loops, there’s not much difference thanks to the internal cache.

## Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE` and a short, one-letter form such as `I`. (If you’re familiar with Perl’s pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is `re.X`, for example.) Multiple flags can be specified by bitwise OR-ing them; `re.I | re.M` sets both the I and M flags, for example.

Here’s a table of the available flags, followed by a more detailed explanation of each one.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII, A</td>
<td>Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property.</td>
</tr>
<tr>
<td>DOTALL, S</td>
<td>Make . match any character, including newlines.</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>IGNORECASE, I</td>
<td>Do case-insensitive matches.</td>
</tr>
<tr>
<td>LOCALE, L</td>
<td>Do a locale-aware match.</td>
</tr>
<tr>
<td>MULTILINE, M</td>
<td>Multi-line matching, affecting ^ and $.</td>
</tr>
<tr>
<td>VERBOSE, X (for ‘extended’)</td>
<td>Enable verbose REs, which can be organized more cleanly and understandably.</td>
</tr>
</tbody>
</table>

**I**

**IGNORECASE**

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, [A-Z] will match lowercase letters, too. Full Unicode matching also works unless the ASCII flag is used to disable non-ASCII matches. When the Unicode patterns [a-z] or [A-Z] are used in combination with the IGNORECASE flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: ‘ı’ (U+0130, Latin capital letter l with dot above), ‘ı’ (U+0131, Latin small letter dotless i), ‘ı’ (U+017F, Latin small letter long s) and ‘K’ (U+212A, Kelvin sign). Spam will match 'Spam', 'spam', 'spAM', or ' sperma' (the latter is matched only in Unicode mode). This lowercasing doesn't take the current locale into account; it will if you also set the LOCALE flag.

**L**

**LOCALE**

Make \w, \W, \b, \B and case-insensitive matching dependent on the current locale instead of the Unicode database.
Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you’re processing encoded French text, you’d want to be able to write \w+ to match words, but \w only matches the character class [A-Za-z] in bytes patterns; it won’t match bytes corresponding to é or ç. If your system is configured properly and a French locale is selected, certain C functions will tell the program that the byte corresponding to é should also be considered a letter. Setting the LOCALE flag when compiling a regular expression will cause the resulting compiled object to use these C functions for \w; this is slower, but also enables \w+ to match French words as you’d expect. The use of this flag is discouraged in Python 3 as the locale mechanism is very unreliable, it only handles one “culture” at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages.

**MULTILINE**

(\^ and \$ haven’t been explained yet; they’ll be introduced in section More Metacharacters.)

Usually \^ matches only at the beginning of the string, and \$ matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, \^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the \$ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).
**DOTALL**

Makes the `.\' special character match any character at all, including a newline; without this flag, `.\' will match anything except a newline.

**ASCII**

Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.

**X**

**VERBOSE**

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. This flag also lets you put comments within a RE that will be ignored by the engine; comments are marked by a `'#\' that's neither in a character class or preceded by an unescaped backslash.

For example, here's a RE that uses `re.VERBOSE`; see how much easier it is to read?

```python
charref = re.compile(r'""'
 &[#]              # Start of a numeric entity
 (                
   \d[0-7]+       # Octal form
 | [0-9]+        # Decimal form
 | x[0-9a-fA-F]+ # Hexadecimal form
 )
```
Without the verbose setting, the RE would look like this:

```python
charref = re.compile("&\#(0[0-7]+"
                     "|[0-9]+"
                     "|x[0-9a-fA-F]+);")
```

In the above example, Python's automatic concatenation of string literals has been used to break up the RE into smaller pieces, but it's still more difficult to understand than the version using `re.VERBOSE`. 
More Pattern Power

So far we’ve only covered a part of the features of regular expressions. In this section, we’ll cover some new metacharacters, and how to use groups to retrieve portions of the text that was matched.

More Metacharacters

There are some metacharacters that we haven’t covered yet. Most of them will be covered in this section.

Some of the remaining metacharacters to be discussed are zero-width assertions. They don’t cause the engine to advance through the string; instead, they consume no characters at all, and simply succeed or fail. For example, \b is an assertion that the current position is located at a word boundary; the position isn’t changed by the \b at all. This means that zero-width assertions should never be repeated, because if they match once at a given location, they can obviously be matched an infinite number of times.

| Alternation, or the “or” operator. If A and B are regular expressions, A|B will match any string that matches either A or B. | has very low precedence in order to make it work reasonably when you’re alternating multi-character strings. Crow|Servo will match either 'Crow' or 'Servo', not 'Cro', a 'w' or an 'S', and 'ervo'.

To match a literal '}', use \}, or enclose it inside a character class, as in [\}].
Matches at the beginning of lines. Unless the MULTILINE flag has been set, this will only match at the beginning of the string. In MULTILINE mode, this also matches immediately after each newline within the string.

For example, if you wish to match the word From only at the beginning of a line, the RE to use is \^From.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

To match a literal \^, use \^\.

$\$

Matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}%', '{block}'))
None
>>> print(re.search('}%', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

To match a literal \$, use \$ or enclose it inside a character class, as in [$].

\A

Matches only at the start of the string. When not in MULTILINE mode, \A and \^ are effectively the same. In MULTILINE mode, they're different: \A still matches only at the beginning of the string,
but ^ may match at any location inside the string that follows a newline character.

\Z
Matches only at the end of the string.

\b
Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

The following example matches class only when it's a complete word; it won't match when it's contained inside another word.

```python
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, \b is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the \b to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous RE, but omits the 'r' in front of the RE string.

```python
>>> p = re.compile(\bclass\b')
>>> print(p.search('no class at all'))
```
Second, inside a character class, where there’s no use for this assertion, \b represents the backspace character, for compatibility with Python’s string literals.

\B

Another zero-width assertion, this is the opposite of \b, only matching when the current position is not at a word boundary.

**Grouping**

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ' : ', like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header’s value.

Groups are marked by the \( ( , ) \) metacharacters. \( ( \) and \( ) \) have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside
them, and you can repeat the contents of a group with a repeating qualifier, such as *, +, ?, or {m, n}. For example, \((ab)^*\) will match zero or more repetitions of ab.

```python
>>> p = re.compile('(ab)*')
>>> print(p.match('ababababab').span())
(0, 10)
```

Groups indicated with \(\(', ')'\) also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so `match object` methods all have group 0 as their default argument. Later we'll see how to express groups that don’t capture the span of text that they match.

```python
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```python
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
```
group() can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```python
>>> m.group(2, 1, 2)
('b', 'abc', 'b')
```

The groups() method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```python
>>> m.groups()
('abc', 'b')
```

Backreferences in a pattern allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, \1 will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise. Remember that Python's string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE.

For example, the following RE detects doubled words in a string.

```python
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren't often useful for just searching through a string — there are few text formats which repeat data in this way — but you'll soon find out that they're very useful when performing string substitutions.
Non-capturing and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 is well known for its powerful additions to standard regular expressions. For these new features the Perl developers couldn't choose new single-keystroke metacharacters or new special sequences beginning with \ without making Perl's regular expressions confusingly different from standard REs. If they chose & as a new metacharacter, for example, old expressions would be assuming that & was a regular character and wouldn't have escaped it by writing \\ or [\&].

The solution chosen by the Perl developers was to use (?...) as the extension syntax. ? immediately after a parenthesis was a syntax error because the ? would have nothing to repeat, so this didn't introduce any compatibility problems. The characters immediately after the ? indicate what extension is being used, so (?=foo) is one thing (a positive lookahead assertion) and (? :foo) is something else (a non-capturing group containing the subexpression foo).

Python supports several of Perl's extensions and adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a P, you know that it's an extension that's specific to Python.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs.
Sometimes you’ll want to use a group to denote a part of a regular expression, but aren’t interested in retrieving the group’s contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can replace the `...` with any other regular expression.

```python
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Except for the fact that you can’t retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as `*`, and nest it within other groups (capturing or non-capturing). `(?:...)` is particularly useful when modifying an existing pattern, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there’s no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other.

A more significant feature is named groups: instead of referring to them by numbers, groups can be referenced by a name.

The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. `name` is, obviously, the name of the group. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The `match object` methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group’s name. Named groups are still given numbers, so you can retrieve information
about a group in two ways:

```python
>>> p = re.compile(r'(?P<word>\b\w+\b)'
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Named groups are handy because they let you use easily-remembered names, instead of having to remember numbers. Here's an example RE from the `imaplib` module:

```python
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>\[123\][0-9])-(?P<mon>\[A-Z\][a-z][a-z])-
    r'(?P<year>\[0-9\][0-9][0-9][0-9])
    r' (?P<hour>\[0-9\][0-9]):(?P<min>\[0-9\][0-9]):(?P<sec>\[0-9\][0-9])
    r' (?P<zonen>[-+])(?P<zoneh>\[0-9\][0-9])(?P<zonem>\[0-9\][0-9])'"')
```

It's obviously much easier to retrieve `m.group('zonem')`, instead of having to remember to retrieve group 9.

The syntax for backreferences in an expression such as `( ... )\1` refers to the number of the group. There's naturally a variant that uses the group name instead of the number. This is another Python extension: `( ?P=name )` indicates that the contents of the group called `name` should again be matched at the current point. The regular expression for finding doubled words, `\b(\w+)\s+\1\b` can also be written as `\b(?P<word>\w+)\s+(?P=word)\b`:

```python
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
```
Lookahead Assertions

Another zero-width assertion is the lookahead assertion. Lookahead assertions are available in both positive and negative form, and look like this:

```
(?=...)
```

Positive lookahead assertion. This succeeds if the contained regular expression, represented here by `...`, successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn’t advance at all; the rest of the pattern is tried right where the assertion started.

```
(?!...)
```

Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression doesn’t match at the current position in the string.

To make this concrete, let’s look at a case where a lookahead is useful. Consider a simple pattern to match a filename and split it apart into a base name and an extension, separated by a `. `. For example, in `news.rc`, `news` is the base name, and `rc` is the filename’s extension.

The pattern to match this is quite simple:

```
.*[.].*$
```

Notice that the `.` needs to be treated specially because it’s a metacharacter, so it’s inside a character class to only match that specific character. Also notice the trailing `$`; this is added to ensure
that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `printers.conf`.

Now, consider complicating the problem a bit; what if you want to match filenames where the extension is not `bat`? Some incorrect attempts:

```regex
.*\.[^b].*$
```

The first attempt above tries to exclude `bat` by requiring that the first character of the extension is not a `b`. This is wrong, because the pattern also doesn’t match `foo.bar`.

```regex
.*\.[^b].|[^a].|[^t]$
```

The expression gets messier when you try to patch up the first solution by requiring one of the following cases to match: the first character of the extension isn’t `b`; the second character isn’t `a`; or the third character isn’t `t`. This accepts `foo.bar` and rejects `autoexec.bat`, but it requires a three-letter extension and won’t accept a filename with a two-letter extension such as `sendmail.cf`. We’ll complicate the pattern again in an effort to fix it.

```regex
.*\.[^b].|[^a].|[^t]$
```

In the third attempt, the second and third letters are all made optional in order to allow matching extensions shorter than three characters, such as `sendmail.cf`.

The pattern’s getting really complicated now, which makes it hard to read and understand. Worse, if the problem changes and you want to exclude both `bat` and `exe` as extensions, the pattern would get even more complicated and confusing.
A negative lookahead cuts through all this confusion:

`.\[(?!bat$)[^.]\]*$` The negative lookahead means: if the expression `bat` doesn’t match at this point, try the rest of the pattern; if `bat$` does match, the whole pattern will fail. The trailing `$` is required to ensure that something like `sample.batch`, where the extension only starts with `bat`, will be allowed. The `[^.]*` makes sure that the pattern works when there are multiple dots in the filename.

Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. The following pattern excludes filenames that end in either `bat` or `exe`:

`.*\[(?!bat$|exe$)[^.]\]*$`
Modifying Strings

Up to this point, we’ve simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>split()</code></td>
<td>Split the string into a list, splitting it wherever the RE matches</td>
</tr>
<tr>
<td><code>sub()</code></td>
<td>Find all substrings where the RE matches, and replace them with a different string</td>
</tr>
<tr>
<td><code>subn()</code></td>
<td>Does the same thing as <code>sub()</code>, but returns the new string and the number of replacements</td>
</tr>
</tbody>
</table>

Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It’s similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; string `split()` only supports splitting by whitespace or by a fixed string. As you’d expect, there’s a module-level `re.split()` function, too.

```
.split(string[, maxsplit=0])
```

Split `string` by the matches of the regular expression. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. If `maxsplit` is nonzero, at most `maxsplit` splits are performed.

You can limit the number of splits made, by passing a value for
maxsplit. When maxsplit is nonzero, at most maxsplit splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter is any sequence of non-alphanumeric characters.

```python
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', '', 'of split().']
```

Sometimes you're not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```python
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'([\W]+)')
>>> p.split('This... is a test. ')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test. ')
['This', '..', 'is', '', 'a', '', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```python
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', '', '', 'words', '', 'words', '.', '', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
```
Search and Replace

Another common task is to find all the matches for a pattern, and replace them with a different string. The sub() method takes a replacement value, which can be either a string or a function, and the string to be processed.

\[
\text{sub}(\text{replacement}, \text{string}[, \text{count}=0])
\]

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in \text{string} by the replacement \text{replacement}. If the pattern isn't found, \text{string} is returned unchanged.

The optional argument \text{count} is the maximum number of pattern occurrences to be replaced; \text{count} must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here’s a simple example of using the sub() method. It replaces colour names with the word colour:

```python
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

The subn() method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```python
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('blue socks and red shoes', 0)
```
Empty matches are replaced only when they're not adjacent to a previous empty match.

If *replacement* is a string, any backslash escapes in it are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \& are left alone. Backreferences, such as \6, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

This example matches the word *section* followed by a string enclosed in {, }, and changes *section* to *subsection*:

There's also a syntax for referring to named groups as defined by the (?P<name>...) syntax. \g<name> will use the substring matched by the group named name, and \g<number> uses the corresponding group number. \g<2> is therefore equivalent to \2, but isn't ambiguous in a replacement string such as \g<2>0. (\20 would be interpreted as a reference to group 20, not a reference to group 2.
followed by the literal character '0'.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```python
g\g<name>
g\g<1>g\g<name>
g\g<name>g\g<1>g\g<name>
g\g<name>g\g<1>g\g<name>
```

replacement can also be a function, which gives you even more control. If replacement is a function, the function is called for every non-overlapping occurrence of pattern. On each call, the function is passed a match object argument for the match and can use this information to compute the desired replacement string and return it.

In the following example, the replacement function translates decimals into hexadecimal:

```python
def hexrepl(match):
    ...  "Return the hex string for a decimal number"
    ...  value = int(match.group())
    ...  return hex(value)
    ...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.'
'Call 0xfff2 for printing, 0xc000 for user code.'
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use
embedded modifiers in the pattern string, e.g. \texttt{sub("(?i)b+", "x", "bbbb BBBB")} returns \texttt{"x x"}.
Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn’t intuitive and at times they don't behave the way you may expect them to. This section will point out some of the most common pitfalls.

Use String Methods

Sometimes using the `re` module is a mistake. If you’re matching a fixed string, or a single character class, and you’re not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they’re usually much faster, because the implementation is a single small C loop that’s been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`’s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character.
You might do this with something like `re.sub(\n', ''', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

In short, before turning to the `re` module, consider whether your problem can be solved with a faster and simpler string method.

**match() versus search()**

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn’t start at zero, `match()` will *not* report it.

```python
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```python
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

Sometimes you'll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a
match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with Crow must match starting with a 'C'. The analysis lets the engine quickly scan through the string looking for the starting character, only trying the full match if a 'C' is found.

Adding .* defeats this optimization, requiring scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use re.search() instead.

**Greedy versus Non-Greedy**

When repeating a regular expression, as in a*, the resulting action is to consume as much of the pattern as possible. This fact often bites you when you're trying to match a pair of balanced delimiters, such as the angle brackets surrounding an HTML tag. The naive pattern for matching a single HTML tag doesn't work because of the greedy nature of .*.

```python
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

The RE matches the '<' in '<html>', and the .* consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the '<' in '<html>' to the '>' in
'</title>', which isn’t what you want.

In this case, the solution is to use the non-greedy qualifiers *?, +?, ??, or \{m,n}\?, which match as little text as possible. In the above example, the ']' is tried immediately after the first '[' matches, and when it fails, the engine advances a character at a time, retrying the ']' at every step. This produces just the right result:

```python
>>> print(re.match('<.*?>', s).group())
<html>
```

(Note that parsing HTML or XML with regular expressions is painful. Quick-and-dirty patterns will handle common cases, but HTML and XML have special cases that will break the obvious regular expression; by the time you've written a regular expression that handles all of the possible cases, the patterns will be very complicated. Use an HTML or XML parser module for such tasks.)

**Using re.VERBOSE**

By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that isn’t inside a character class is ignored. This means
that an expression such as dog | cat is equivalent to the less readable dog|cat, but \[a b\] will still match the characters 'a', 'b', or a space. In addition, you can also put comments inside a RE; comments extend from a # character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```python
pat = re.compile(r""
   \s*                      # Skip leading whitespace
   (?P<header>[^:]+)       # Header name
   \s* :                   # Whitespace, and a colon
   (?P<value>.*)          # The header's value -- *? used
   \s* $                  # lose the following trailing whitespace
   """, re.VERBOSE)
```

This is far more readable than:

```python
pat = re.compile(r"\s* (?P<header>[^:]+) \s* : (?P<value>.*?) \s* $", re.VERBOSE)
```
Feedback

Regular expressions are a complicated topic. Did this document help you understand them? Were there parts that were unclear, or Problems you encountered that weren't covered here? If so, please send suggestions for improvements to the author.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's Mastering Regular Expressions, published by O'Reilly. Unfortunately, it exclusively concentrates on Perl and Java’s flavours of regular expressions, and doesn't contain any Python material at all, so it won't be useful as a reference for programming in Python. (The first edition covered Python’s now-removed regex module, which won't help you much.) Consider checking it out from your library.
Socket Programming HOWTO

**Author:** Gordon McMillan

**Abstract**

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It’s not really a tutorial - you’ll still have work to do in getting things operational. It doesn’t cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.
Sockets

I'm only going to talk about INET (i.e. IPv4) sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM (i.e. TCP) sockets - unless you really know what you're doing (in which case this HOWTO isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that “socket” can mean a number of subtly different things, depending on context. So first, let's make a distinction between a “client” socket - an endpoint of a conversation, and a “server” socket, which is more like a switchboard operator. The client application (your browser, for example) uses “client” sockets exclusively; the web server it's talking to uses both “server” sockets and “client” sockets.

History

Of the various forms of IPC, sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the Internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other
schemes).
Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(('www.python.org', 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a "server socket":

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a "server" socket, but one that was only visible within the same machine. `s.bind(('', 80))` specifies that
the socket is reachable by any address the machine happens to have.

A second thing to note: low number ports are usually reserved for “well known” services (HTTP, SNMP etc). If you’re playing around, use a nice high number (4 digits).

Finally, the argument to listen tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a “server” socket, listening on port 80, we can enter the mainloop of the web server:

```python
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle clientsocket, create a new process to handle clientsocket, or restructure this app to use non-blocking sockets, and multiplex between our “server” socket and any active clientsockets using select. More about that later. The important thing to understand now is this: this is all a “server” socket does. It doesn’t send any data. It doesn’t receive any data. It just produces “client” sockets. Each clientsocket is created in response to some other “client” socket doing a connect() to the host and port we're bound to. As soon as we've created that clientsocket, we go back to listening for more connections. The two “clients” are free to chat it
up - they are using some dynamically allocated port which will be recycled when the conversation ends.

**IPC**

If you need fast IPC between two processes on one machine, you should look into pipes or shared memory. If you do decide to use AF_INET sockets, bind the “server” socket to 'localhost'. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

**See also:** The *multiprocessing* integrates cross-platform IPC into a higher-level API.
Using a Socket

The first thing to note, is that the web browser’s “client” socket and the web server’s “client” socket are identical beasts. That is, this is a “peer to peer” conversation. Or to put it another way, as the designer, you will have to decide what the rules of etiquette are for a conversation. Normally, the connecting socket starts the conversation, by sending in a request, or perhaps a signon. But that’s a design decision - it’s not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use send and recv, or you can transform your client socket into a file-like beast and use read and write. The latter is the way Java presents its sockets. I’m not going to talk about it here, except to warn you that you need to use flush on sockets. These are buffered “files”, and a common mistake is to write something, and then read for a reply. Without a flush in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - send and recv operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (send) or emptied (recv). They then tell you how many bytes they handled. It is your responsibility to call them again until your message has been completely dealt with.

When a recv returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data
successfully; I'll talk more about this later.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that there is no \texttt{EOT} on a socket. I repeat: if a socket \texttt{send} or \texttt{recv} returns after handling 0 bytes, the connection has been broken. If the connection has not been broken, you may wait on a \texttt{recv} forever, because the socket will not tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: messages must either be fixed length (yuck), or be delimited (shrug), or indicate how long they are (much better), or end by shutting down the connection. The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```python
class MySocket:
    """demonstration class only
    - coded for clarity, not efficiency
    ""

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
```

```python
```
self.sock.connect((host, port))

def mysend(self, msg):
    totalsent = 0
    while totalsent < MSGLEN:
        sent = self.sock.send(msg[totalsent:])
        if sent == 0:
            raise RuntimeError("socket connection broken")
        totalsent = totalsent + sent

def myreceive(self):
    chunks = []
    bytes_recd = 0
    while bytes_recd < MSGLEN:
        chunk = self.sock.recv(min(MSGLEN - bytes_recd, 1024))
        if chunk == b'':
            raise RuntimeError("socket connection broken")
        chunks.append(chunk)
        bytes_recd = bytes_recd + len(chunk)
    return b''.join(chunks)

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded \0 characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen()` if the message has embedded \0s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've
received for a delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You’ll need to put that aside and hold onto it, until it’s needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you’ll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you’ll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

**Binary Data**

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, a Motorola chip will represent a 16 bit integer with the value 1 as the two hex bytes 00 01. Intel and DEC, however, are byte-reversed - that same 1 is 01 00. Socket libraries have calls for converting 16 and 32 bit integers - `ntohl`, `htonl`, `ntohs`, `htons` where “n” means `network` and “h” means `host`, “s” means `short` and “l” means `long`. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around
appropriately.

In these days of 32 bit machines, the ascii representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, all those longs have the value 0, or maybe 1. The string “0” would be two bytes, while binary is four. Of course, this doesn’t fit well with fixed-length messages. Decisions, decisions.
Disconnecting

Strictly speaking, you’re supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean “I’m not going to send anymore, but I’ll still listen”, or “I’m not listening, good riddance!”. Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server “This client is done sending, but can still receive.” The server can detect “EOF” by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it’s needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you’re just being slow. Please `close` your sockets when you’re done.

When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. TCP is a reliable protocol, and it will wait a
long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do not try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.
Non-blocking Sockets

If you’ve understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You’ll still use the same calls, in much the same ways. It’s just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(0)` to make it non-blocking. In C, it's more complex, (for one thing, you’ll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable Posix flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it’s the exact same idea. You do this after creating the socket, but before using it. (Actually, if you’re nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don’t believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let’s skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

```python
ready_to_read, ready_to_write, in_error = \
    select.select(\n        potential_readers,\n        potential_writers,\n```
You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list. You’ll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a “server” socket, put it in the `potential_readers` list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the `potential_writers` list. If it shows up in the writable list, you have a decent chance that it has connected.

Actually, `select` can be handy even with blocking sockets. It’s one way of determining whether you will block - the socket returns as readable when there’s something in the buffers. However, this still doesn’t help with the problem of determining whether the other end is done, or just busy with something else.
Portability alert: On Unix, `select` works both with the sockets and files. Don’t try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets.
Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.
Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```python
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method. It modifies the list in-place (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

```python
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```python
[1, 2, 3, 4, 5]
```
Key Functions

Both `list.sort()` and `sorted()` have a `key` parameter to specify a function to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

```python
>>> sorted("This is a test string from Andrew".split)[
'a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the `key` parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```python
>>> student_tuples = [
... ('john', 'A', 15),
... ('jane', 'B', 12),
... ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The same technique works for objects with named attributes. For example:

```python
>>> class Student:
...     def __init__(self, name, grade, age):
... ```
...   self.name = name
...   self.grade = grade
...   self.age = age
...   def __repr__(self):
...       return repr((self.name, self.grade, self.age))

>>> student_objects = [  
...     Student('john', 'A', 15),  
...     Student('jane', 'B', 12),  
...     Student('dave', 'B', 10),  
... ]

>>> sorted(student_objects, key=lambda student: student[2])
[(('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15))]
Operator Module Functions

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The operator module has itemgetter(), attrgetter(), and a methodcaller() function.

Using those functions, the above examples become simpler and faster:

```python
>>> from operator import itemgetter, attrgetter

>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

The operator module functions allow multiple levels of sorting. For example, to sort by grade then by age:

```python
>>> sorted(student_tuples, key=itemgetter(1, 2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
Ascending and Descending

Both `list.sort()` and `sorted()` accept a `reverse` parameter with a boolean value. This is used to flag descending sorts. For example, to get the student data in reverse `age` order:

```python
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

```python
>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```
Sort Stability and Complex Sorts

Sorts are guaranteed to be **stable**. That means that when multiple records have the same key, their original order is preserved.

```python
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2), ('red', 2)]
```

```python
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Notice how the two records for `blue` retain their original order so that `('blue', 1)` is guaranteed to precede `('blue', 2)`.

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending `grade` and then ascending `age`, do the `age` sort first and then sort again using `grade`:

```python
>>> s = sorted(student_objects, key=attrgetter('age'))
```

```python
>>> sorted(s, key=attrgetter('grade'), reverse=True)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The **Timsort** algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.
The Old Way Using Decorate-Sort-Undecorate

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

For example, to sort the student data by grade using the DSU approach:

```python
>>> decorated = [(student.grade, i, student) for i, student in decorated]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index $i$ in the decorated list, but including it gives two benefits:

- The sort is stable – if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain complex numbers which cannot be sorted directly.
Another name for this idiom is **Schwartzian transform**, after Randal L. Schwartz, who popularized it among Perl programmers.

Now that Python sorting provides key-functions, this technique is not often needed.
The Old Way Using the `cmp` Parameter

Many constructs given in this HOWTO assume Python 2.4 or later. Before that, there was no `sorted()` builtin and `list.sort()` took no keyword arguments. Instead, all of the Py2.x versions supported a `cmp` parameter to handle user specified comparison functions.

In Py3.0, the `cmp` parameter was removed entirely (as part of a larger effort to simplify and unify the language, eliminating the conflict between rich comparisons and the `__cmp__()` magic method).

In Py2.x, `sort` allowed an optional function which can be called for doing the comparisons. That function should take two arguments to be compared and then return a negative value for less-than, return zero if they are equal, or return a positive value for greater-than. For example, we can do:

```python
>>> def numeric_compare(x, y):
...   return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare)  # doctest: +SKIP
[1, 2, 3, 4, 5]
```

Or you can reverse the order of comparison with:

```python
>>> def reverse_numeric(x, y):
...   return y - x
>>> sorted([5, 2, 4, 1, 3], cmp=reverse_numeric)  # doctest: +SKIP
[5, 4, 3, 2, 1]
```

When porting code from Python 2.x to 3.x, the situation can arise when you have the user supplying a comparison function and you need to convert that to a key function. The following wrapper makes
that easy to do:

```python
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'

class K:
    def __init__(self, obj, *args):
        self.obj = obj
    def __lt__(self, other):
        return mycmp(self.obj, other.obj) < 0
    def __gt__(self, other):
        return mycmp(self.obj, other.obj) > 0
    def __eq__(self, other):
        return mycmp(self.obj, other.obj) == 0
    def __le__(self, other):
        return mycmp(self.obj, other.obj) <= 0
    def __ge__(self, other):
        return mycmp(self.obj, other.obj) >= 0
    def __ne__(self, other):
        return mycmp(self.obj, other.obj) != 0

return K
```

To convert to a key function, just wrap the old comparison function:

```python
>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]
```

In Python 3.2, the `functools.cmp_to_key()` function was added to the `functools` module in the standard library.
Odd and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function.

- The `reverse` parameter still maintains sort stability (so that records with equal keys retain the original order). Interestingly, that effect can be simulated without the parameter by using the builtin `reversed()` function twice:

  ```python
  >>> data = [('red', 1), ('blue', 1), ('red', 2),
  >>>          ('red', 1), ('blue', 1), ('blue', 2),
  >>>          ('red', 2), ('blue', 1), ('blue', 2),
  >>>          ('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
  >>> standard_way = sorted(data, key=itemgetter(0), reverse=True)
  >>> double_reversed = list(reversed(sorted(reversed(data))))
  >>> assert standard_way == double_reversed
  >>> standard_way
  [('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
  ```

- The sort routines are guaranteed to use `__lt__()` when making comparisons between two objects. So, it is easy to add a standard sort order to a class by defining an `__lt__()` method:

  ```python
  >>> Student.__lt__ = lambda self, other: self.age < other.age
  >>> sorted(student_objects)
  [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
  ```

- Key functions need not depend directly on the objects being sorted. A key function can also access external resources. For instance, if the student grades are stored in a dictionary, they can be used to sort a separate list of student names:

  ```python
  >>> students = ['dave', 'john', 'jane']
  >>> newgrades = {'john': 'F', 'jane': 'A', 'dave': 'B'}
  >>> sorted(students, key=newgrades.get)
  ['john', 'jane', 'dave']
  ```
```python
>>> sorted(students, key=newgrades.__getitem__)
['jane', 'dave', 'john']
```
Unicode HOWTO

Release: 1.12

This HOWTO discusses Python’s support for the Unicode specification for representing textual data, and explains various problems that people commonly encounter when trying to work with Unicode.
Introduction to Unicode

Definitions

Today's programs need to be able to handle a wide variety of characters. Applications are often internationalized to display messages and output in a variety of user-selectable languages; the same program might need to output an error message in English, French, Japanese, Hebrew, or Russian. Web content can be written in any of these languages and can also include a variety of emoji symbols. Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all these different possible characters.

Unicode (https://www.unicode.org/) is a specification that aims to list every character used by human languages and give each character its own unique code. The Unicode specifications are continually revised and updated to add new languages and symbols.

A character is the smallest possible component of a text. ‘A’, ‘B’, ‘C’, etc., are all different characters. So are ‘È’ and ‘Í’. Characters vary depending on the language or context you're talking about. For example, there's a character for “Roman Numeral One”, ‘Ⅰ’, that's separate from the uppercase letter ‘I’. They'll usually look the same, but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by code points. A code point value is an integer in the range 0 to 0x10FFFF (about 1.1 million values, with some 110 thousand assigned so far). In the standard and in this document, a code point is written using the notation U+265E to mean the character with value 0x265e (9,822 in decimal).
The Unicode standard contains a lot of tables listing characters and their corresponding code points:

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0061</td>
<td>'a'</td>
<td>LATIN SMALL LETTER A</td>
</tr>
<tr>
<td>0062</td>
<td>'b'</td>
<td>LATIN SMALL LETTER B</td>
</tr>
<tr>
<td>0063</td>
<td>'c'</td>
<td>LATIN SMALL LETTER C</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>007B</td>
<td>'{'</td>
<td>LEFT CURLY BRACKET</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2167</td>
<td>'Ⅶ'</td>
<td>ROMAN NUMERAL EIGHT</td>
</tr>
<tr>
<td>2168</td>
<td>'Ⅸ'</td>
<td>ROMAN NUMERAL NINE</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>265E</td>
<td>'♞'</td>
<td>BLACK CHESS KNIGHT</td>
</tr>
<tr>
<td>265F</td>
<td>'♟'</td>
<td>BLACK CHESS PAWN</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1F600</td>
<td>'ᛡ'</td>
<td>GRINNING FACE</td>
</tr>
<tr>
<td>1F609</td>
<td>'ᛪ'</td>
<td>WINKING FACE</td>
</tr>
</tbody>
</table>

Strictly, these definitions imply that it's meaningless to say ‘this is character U+265E’. U+265E is a code point, which represents some particular character; in this case, it represents the character ‘BLACK CHESS KNIGHT’, ‘♞’. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

A character is represented on a screen or on paper by a set of graphical elements that's called a **glyph**. The glyph for an uppercase A, for example, is two diagonal strokes and a horizontal stroke, though the exact details will depend on the font being used. Most Python code doesn't need to worry about glyphs; figuring out the correct glyph to display is generally the job of a GUI toolkit or a terminal's font renderer.

**Encodings**
To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 through \(0x10FFFF\) (1,114,111 decimal). This sequence of code points needs to be represented in memory as a set of **code units**, and **code units** are then mapped to 8-bit bytes. The rules for translating a Unicode string into a sequence of bytes are called a **character encoding**, or just an **encoding**.

The first encoding you might think of is using 32-bit integers as the code unit, and then using the CPU's representation of 32-bit integers. In this representation, the string “Python” might look like this:

<table>
<thead>
<tr>
<th>P</th>
<th>y</th>
<th>t</th>
<th>h</th>
<th>o</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x50</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>79</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

This representation is straightforward but using it presents a number of problems.

1. It's not portable; different processors order the bytes differently.
2. It's very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255, so a lot of space is occupied by \(0x00\) bytes. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn't matter too much (desktop computers have gigabytes of RAM, and strings aren't usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
3. It's not compatible with existing C functions such as `strlen()`, so a new family of wide string functions would need to be used.

Therefore this encoding isn't used very much, and people instead choose other encodings that are more efficient and convenient, such as UTF-8.
UTF-8 is one of the most commonly used encodings, and Python often defaults to using it. UTF stands for “Unicode Transformation Format”, and the ‘8’ means that 8-bit values are used in the encoding. (There are also UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules:

1. If the code point is < 128, it’s represented by the corresponding byte value.
2. If the code point is >= 128, it’s turned into a sequence of two, three, or four bytes, where each byte of the sequence is between 128 and 255.

UTF-8 has several convenient properties:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a sequence of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can’t handle zero bytes.
3. A string of ASCII text is also valid UTF-8 text.
4. UTF-8 is fairly compact; the majority of commonly used characters can be represented with one or two bytes.
5. If bytes are corrupted or lost, it’s possible to determine the start of the next UTF-8-encoded code point and resynchronize. It’s also unlikely that random 8-bit data will look like valid UTF-8.

References

The Unicode Consortium site has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. A chronology of the origin and development of Unicode is also available on the site.
On the Computerphile Youtube channel, Tom Scott briefly discusses the history of Unicode and UTF-8 <https://www.youtube.com/watch?v=MijmeoH9LT4> (9 minutes 36 seconds).

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables.

Another good introductory article was written by Joel Spolsky. If this introduction didn’t make things clear to you, you should try reading this alternate article before continuing.

Wikipedia entries are often helpful; see the entries for “character encoding” and UTF-8, for example.
Python’s Unicode Support

Now that you’ve learned the rudiments of Unicode, we can look at Python’s Unicode features.

The String Type

Since Python 3.0, the language’s `str` type contains Unicode characters, meaning any string created using "unicode rocks!", 'unicode rocks!', or the triple-quoted string syntax is stored as Unicode.

The default encoding for Python source code is UTF-8, so you can simply include a Unicode character in a string literal:

```python
try:
    with open('/tmp/input.txt', 'r') as f:
        ...
except OSError:
    # 'File not found' error message.
    print("Fichier non trouvé")
```

Side note: Python 3 also supports using Unicode characters in identifiers:

```python
répertoire = "/tmp/records.log"
with open(répertoire, "w") as f:
    f.write("test\n")
```

If you can’t enter a particular character in your editor or want to keep the source code ASCII-only for some reason, you can also use escape sequences in string literals. (Depending on your system, you may see
the actual capital-delta glyph instead of a u escape.)

```
>>> "\N{GREEK CAPITAL LETTER DELTA}"  # Using the character name
'\u0394'
>>> "\u0394"                         # Using a 16-bit hex value
'\u0394'
>>> "\U00000394"                     # Using a 32-bit hex value
'\u0394'
```

In addition, one can create a string using the `decode()` method of `bytes`. This method takes an `encoding` argument, such as `UTF-8`, and optionally an `errors` argument.

The `errors` argument specifies the response when the input string can't be converted according to the encoding's rules. Legal values for this argument are `strict` (raise a `UnicodeDecodeError` exception), `replace` (use U+FFFD, REPLACEMENT CHARACTER), `ignore` (just leave the character out of the Unicode result), or `backslashreplace` (inserts a `\xNN` escape sequence). The following examples show the differences:

```
>>> b'\x80abc'.decode("utf-8", "strict")
Traceback (most recent call last):
  ...  
UnicodeDecodeError: 'utf-8' codec can't decode byte invalid start byte
>>> b'\x80abc'.decode("utf-8", "replace")
'\ufffdabc'
>>> b'\x80abc'.decode("utf-8", "backslashreplace")
'\\x80abc'
>>> b'\x80abc'.decode("utf-8", "ignore")
'abc'
```
Encodings are specified as strings containing the encoding’s name. Python comes with roughly 100 different encodings; see the Python Library Reference at Standard Encodings for a list. Some encodings have multiple names; for example, 'latin-1', 'iso_8859_1' and '8859' are all synonyms for the same encoding.

One-character Unicode strings can also be created with the chr() built-in function, which takes integers and returns a Unicode string of length 1 that contains the corresponding code point. The reverse operation is the built-in ord() function that takes a one-character Unicode string and returns the code point value:

```
>>> chr(57344)
'\ue000'
>>> ord('\ue000')
57344
```

Converting to Bytes

The opposite method of bytes.decode() is str.encode(), which returns a bytes representation of the Unicode string, encoded in the requested encoding.

The errors parameter is the same as the parameter of the decode() method but supports a few more possible handlers. As well as 'strict', 'ignore', and 'replace' (which in this case inserts a question mark instead of the unencodable character), there is also 'xmlcharrefreplace' (inserts an XML character reference), backslashreplace (inserts a \uNNNN escape sequence) and namereplace (inserts a \N{...} escape sequence).

The following example shows the different results:
u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xe8\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character ' ' in position 0: ordinal not in range(128)

The low-level routines for registering and accessing the available encodings are found in the codecs module. Implementing new encodings also requires understanding the codecs module. However, the encoding and decoding functions returned by this module are usually more low-level than is comfortable, and writing new encodings is a specialized task, so the module won't be covered in this HOWTO.

Unicode Literals in Python Source Code

In Python source code, specific Unicode code points can be written using the \u escape sequence, which is followed by four hex digits giving the code point. The \U escape sequence is similar, but expects eight hex digits, not four:
Using escape sequences for code points greater than 127 is fine in small doses, but becomes an annoyance if you're using many accented characters, as you would in a program with messages in French or some other accent-using language. You can also assemble strings using the `chr()` built-in function, but this is even more tedious.

Ideally, you’d want to be able to write literals in your language’s natural encoding. You could then edit Python source code with your favorite editor which would display the accented characters naturally, and have the right characters used at runtime.

Python supports writing source code in UTF-8 by default, but you can use almost any encoding if you declare the encoding being used. This is done by including a special comment as either the first or second line of the source file:

```bash
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = 'abcdé'
print(ord(u[-1]))
```

The syntax is inspired by Emacs's notation for specifying variables local to a file. Emacs supports many different variables, but Python only supports ‘coding’. The `-*-` symbols indicate to Emacs that the comment is special; they have no significance to Python but are a
convention. Python looks for `coding: name` or `coding=name` in the comment.

If you don’t include such a comment, the default encoding used will be UTF-8 as already mentioned. See also PEP 263 for more information.

Unicode Properties

The Unicode specification includes a database of information about code points. For each defined code point, the information includes the character’s name, its category, the numeric value if applicable (for characters representing numeric concepts such as the Roman numerals, fractions such as one-third and four-fifths, etc.). There are also display-related properties, such as how to use the code point in bidirectional text.

The following program displays some information about several characters, and prints the numeric value of one particular character:

```python
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(3972) + chr(6000) +

for i, c in enumerate(u):
    print(i, '%04x' % ord(c), unicodedata.category(c),
          unicodedata.name(c))

# Get numeric value of second character
print(unicodedata.numeric(u[1]))
```

When run, this prints:

```
0 00e9 L1 LATIN SMALL LETTER E WITH ACUTE
```
The category codes are abbreviations describing the nature of the character. These are grouped into categories such as “Letter”, “Number”, “Punctuation”, or “Symbol”, which in turn are broken up into subcategories. To take the codes from the above output, 'Ll' means ‘Letter, lowercase’, 'No' means “Number, other”, 'Mn' is “Mark, nonspacing”, and 'So' is “Symbol, other”. See the General Category Values section of the Unicode Character Database documentation for a list of category codes.

Comparing Strings

Unicode adds some complication to comparing strings, because the same set of characters can be represented by different sequences of code points. For example, a letter like ‘é’ can be represented as a single code point U+00EA, or as U+0065 U+0302, which is the code point for ‘e’ followed by a code point for ‘COMBINING CIRCUMFLEX ACCENT’. These will produce the same output when printed, but one is a string of length 1 and the other is of length 2.

One tool for a case-insensitive comparison is the casefold() string method that converts a string to a case-insensitive form following an algorithm described by the Unicode Standard. This algorithm has special handling for characters such as the German letter ‘ß’ (code point U+00DF), which becomes the pair of lowercase letters ‘ss’.

```python
>>> street = 'Gürzenichstraße'
>>> street.casefold()
```
A second tool is the `unicodedata` module's `normalize()` function that converts strings to one of several normal forms, where letters followed by a combining character are replaced with single characters. `normalize()` can be used to perform string comparisons that won't falsely report inequality if two strings use combining characters differently:

```python
import unicodedata

def compare_strs(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(s1) == NFD(s2)

single_char = 'ê'
multiple_chars = '\N{LATIN SMALL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}

print('length of first string=', len(single_char))
print('length of second string=', len(multiple_chars))
print(compare_strs(single_char, multiple_chars))
```

When run, this outputs:

```
$ python3 compare-strs.py
length of first string= 1
length of second string= 2
True
```

The first argument to the `normalize()` function is a string giving the desired normalization form, which can be one of ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’.
The Unicode Standard also specifies how to do caseless comparisons:

```python
import unicodedata

def compare_caseless(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(NFD(s1).casefold()) == NFD(NFD(s2).casefold())

# Example usage
single_char = 'ê'
multiple_chars = '\N{LATIN CAPITAL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'

print(compare_caseless(single_char, multiple_chars))
```

This will print `True`. (Why is NFD() invoked twice? Because there are a few characters that make casefold() return a non-normalized string, so the result needs to be normalized again. See section 3.13 of the Unicode Standard for a discussion and an example.)

**Unicode Regular Expressions**

The regular expressions supported by the `re` module can be provided either as bytes or strings. Some of the special character sequences such as `\d` and `\w` have different meanings depending on whether the pattern is supplied as bytes or a string. For example, `\d` will match the characters `[0-9]` in bytes but in strings will match any character that's in the 'Nd' category.

The string in this example has the number 57 written in both Thai and Arabic numerals:
```python
import re
p = re.compile(r'\d+')
s = "Over \u0e55\u0e57 57 flavours"
m = p.search(s)
print(repr(m.group()))
```

When executed, \d+ will match the Thai numerals and print them out. If you supply the `re.ASCII` flag to `compile()`, \d+ will match the substring “57” instead.

Similarly, \w matches a wide variety of Unicode characters but only [a-zA-Z0-9_] in bytes or if `re.ASCII` is supplied, and \s will match either Unicode whitespace characters or [\t\n\r\f\v].

References

Some good alternative discussions of Python’s Unicode support are:

- [Processing Text Files in Python 3](https://docs.python.org/3/library/str.html), by Nick Coghlan.
- [Pragmatic Unicode](https://www.python.org/ devs/more/unicode/), a PyCon 2012 presentation by Ned Batchelder.

The `str` type is described in the Python library reference at [Text Sequence Type — `str`](https://docs.python.org/3/library/str.html).

The documentation for the `unicodedata` module.

The documentation for the `codecs` module.

Marc-André Lemburg gave a presentation titled “Python and Unicode” ([PDF slides](https://europython.org/slides/2002/lemburg_5 만들기) at EuroPython 2002. The slides are an excellent overview of the design of Python 2’s Unicode features (where the Unicode string
type is called `unicode` and literals start with `u`).
Reading and Writing Unicode Data

Once you've written some code that works with Unicode data, the next problem is input/output. How do you get Unicode strings into your program, and how do you convert Unicode into a form suitable for storage or transmission?

It's possible that you may not need to do anything depending on your input sources and output destinations; you should check whether the libraries used in your application support Unicode natively. XML parsers often return Unicode data, for example. Many relational databases also support Unicode-valued columns and can return Unicode values from an SQL query.

Unicode data is usually converted to a particular encoding before it gets written to disk or sent over a socket. It's possible to do all the work yourself: open a file, read an 8-bit bytes object from it, and convert the bytes with `bytes.decode(encoding)`. However, the manual approach is not recommended.

One problem is the multi-byte nature of encodings; one Unicode character can be represented by several bytes. If you want to read the file in arbitrary-sized chunks (say, 1024 or 4096 bytes), you need to write error-handling code to catch the case where only part of the bytes encoding a single Unicode character are read at the end of a chunk. One solution would be to read the entire file into memory and then perform the decoding, but that prevents you from working with files that are extremely large; if you need to read a 2 GiB file, you need 2 GiB of RAM. (More, really, since for at least a moment you'd need to have both the encoded string and its Unicode version in memory.)

The solution would be to use the low-level decoding interface to catch
the case of partial coding sequences. The work of implementing this has already been done for you: the built-in open() function can return a file-like object that assumes the file's contents are in a specified encoding and accepts Unicode parameters for methods such as read() and write(). This works through open()’s encoding and errors parameters which are interpreted just like those in str.encode() and bytes.decode().

Reading Unicode from a file is therefore simple:

```python
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

It’s also possible to open files in update mode, allowing both reading and writing:

```python
with open('test', encoding='utf-8', mode='w') as f:
    f.write('\u4500 blah blah blah
')
    f.seek(0)
    print(repr(f.readline()[:1]))
```

The Unicode character U+FEFF is used as a byte-order mark (BOM), and is often written as the first character of a file in order to assist with autodetection of the file's byte ordering. Some encodings, such as UTF-16, expect a BOM to be present at the start of a file; when such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as ‘utf-16-le’ and ‘utf-16-be’ for little-endian and big-endian encodings, that specify one particular byte ordering and don’t skip the BOM.

In some areas, it is also convention to use a “BOM” at the start of
UTF-8 encoded files; the name is misleading since UTF-8 is not byte-order dependent. The mark simply announces that the file is encoded in UTF-8. For reading such files, use the ‘utf-8-sig’ codec to automatically skip the mark if present.

Unicode filenames

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. Today Python is converging on using UTF-8: Python on MacOS has used UTF-8 for several versions, and Python 3.6 switched to using UTF-8 on Windows as well. On Unix systems, there will only be a filesystem encoding if you’ve set the LANG or LC_CTYPE environment variables; if you haven’t, the default encoding is again UTF-8.

The sys.getfilesystemencoding() function returns the encoding to use on your current system, in case you want to do the encoding manually, but there’s not much reason to bother. When opening a file for reading or writing, you can usually just provide the Unicode string as the filename, and it will be automatically converted to the right encoding for you:

```python
filename = 'filename\u4500abc'
with open(filename, 'w') as f:
    f.write('blah
')
```

Functions in the os module such as os.stat() will also accept Unicode filenames.

The os.listdir() function returns filenames, which raises an
issue: should it return the Unicode version of filenames, or should it return bytes containing the encoded versions? `os.listdir()` can do both, depending on whether you provided the directory path as bytes or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem’s encoding and a list of Unicode strings will be returned, while passing a byte path will return the filenames as bytes. For example, assuming the default filesystem encoding is UTF-8, running the following program:

```python
fn = 'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print(os.listdir(b'.'))
print(os.listdir('.'))
```

will produce the following output:

```
$ python listdir-test.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

The first list contains UTF-8-encoded filenames, and the second list contains the Unicode versions.

Note that on most occasions, you should can just stick with using Unicode with these APIs. The bytes APIs should only be used on systems where undecodable file names can be present; that’s pretty much only Unix systems now.

**Tips for Writing Unicode-aware Programs**
This section provides some suggestions on writing software that deals with Unicode.

The most important tip is:

   Software should only work with Unicode strings internally, decoding the input data as soon as possible and encoding the output only at the end.

If you attempt to write processing functions that accept both Unicode and byte strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. There is no automatic encoding or decoding: if you do e.g. `str + bytes`, a `TypeError` will be raised.

When using data coming from a web browser or some other untrusted source, a common technique is to check for illegal characters in a string before using the string in a generated command line or storing it in a database. If you're doing this, be careful to check the decoded string, not the encoded bytes data; some encodings may have interesting properties, such as not being bijective or not being fully ASCII-compatible. This is especially true if the input data also specifies the encoding, since the attacker can then choose a clever way to hide malicious text in the encoded bytestream.

Converting Between File Encodings

The `StreamRecoder` class can transparently convert between encodings, taking a stream that returns data in encoding #1 and behaving like a stream returning data in encoding #2.

For example, if you have an input file `f` that's in Latin-1, you can wrap it with a `StreamRecoder` to return bytes encoded in UTF-8:
new_f = codecs.StreamRecoder(f,
    # en/decoder: used by read() to encode its results
    # by write() to decode its input.
    codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'))

    # reader/writer: used to read and write to the stream
    codecs.getreader('latin-1'), codecs.getwriter('latin-1'))

Files in an Unknown Encoding

What can you do if you need to make a change to a file, but don't know the file's encoding? If you know the encoding is ASCII-compatible and only want to examine or modify the ASCII parts, you can open the file with the surrogateescape error handler:

```
with open(fname, 'r', encoding="ascii", errors="surrogateescape")
data = f.read()

# make changes to the string 'data'
with open(fname + '.new', 'w',
    encoding="ascii", errors="surrogateescape")
f.write(data)
```

The surrogateescape error handler will decode any non-ASCII bytes as code points in a special range running from U+DC80 to U+DCFF. These code points will then turn back into the same bytes when the surrogateescape error handler is used to encode the data and write it back out.

References
One section of *Mastering Python 3 Input/Output*, a PyCon 2010 talk by David Beazley, discusses text processing and binary data handling.

The PDF slides for Marc-André Lemburg’s presentation “Writing Unicode-aware Applications in Python” discuss questions of character encodings as well as how to internationalize and localize an application. These slides cover Python 2.x only.

*The Guts of Unicode in Python* is a PyCon 2013 talk by Benjamin Peterson that discusses the internal Unicode representation in Python 3.3.
Acknowledgements

The initial draft of this document was written by Andrew Kuchling. It has since been revised further by Alexander Belopolsky, Georg Brandl, Andrew Kuchling, and Ezio Melotti.

Thanks to the following people who have noted errors or offered suggestions on this article: Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Serhiy Storchaka, Eryk Sun, Chad Whitacre, Graham Wideman.
HOWTO Fetch Internet Resources Using The urllib Package

**Author:** Michael Foord

**Note:** There is a French translation of an earlier revision of this HOWTO, available at urllib2 - Le Manuel manquant.
Introduction

`urllib.request` is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the `urlopen` function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

`urllib.request` supports fetching URLs for many “URL schemes” (identified by the string before the `:` in URL - for example `ftp` is the URL scheme of `ftp://python.org/`) using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.

For straightforward situations `urlopen` is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol. The most comprehensive and authoritative reference to HTTP is RFC 2616. This is a technical document and not intended to be easy to read. This HOWTO aims to illustrate using `urllib`, with enough detail about HTTP to help you through. It is not intended to replace the `urllib.request` docs, but is supplementary to them.
Fetching URLs

The simplest way to use urllib.request is as follows:

```python
import urllib.request
with urllib.request.urlopen('http://python.org/') as html:
    html = response.read()
```

If you wish to retrieve a resource via URL and store it in a temporary location, you can do so via the `shutil.copyfileobj()` and `tempfile.NamedTemporaryFile()` functions:

```python
import shutil
import tempfile
import urllib.request

with urllib.request.urlopen('http://python.org/') as html:
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        shutil.copyfileobj(response, tmp_file)

with open(tmp_file.name) as html:
    pass
```

Many uses of urllib will be that simple (note that instead of an ‘http:’ URL we could have used a URL starting with ‘ftp:’, ‘file:’, etc.). However, it’s the purpose of this tutorial to explain the more complicated cases, concentrating on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. urllib.request mirrors this with a `Request` object which represents the HTTP request you are making. In its simplest form you create a `Request` object that specifies the URL
you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `.read()` on the response:

```python
import urllib.request

req = urllib.request.Request('http://www.voidspace.org.uk')
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

Note that `urllib.request` makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

```python
req = urllib.request.Request('ftp://example.com/')
```

In the case of HTTP, there are two extra things that Request objects allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information (“metadata”) about the data or the about request itself, to the server - this information is sent as HTTP “headers”. Let's look at each of these in turn.

**Data**

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script or other web application). With HTTP, this is often done using what’s known as a **POST** request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case of HTML forms, the data needs to be encoded in
a standard way, and then passed to the Request object as the data argument. The encoding is done using a function from the urllib.parse library.

```python
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python'}

data = urllib.parse.urlencode(values)
data = data.encode('ascii')  # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

Note that other encodings are sometimes required (e.g. for file upload from HTML forms - see HTML Specification, Form Submission for more details).

If you do not pass the data argument, urllib uses a GET request. One way in which GET and POST requests differ is that POST requests often have “side-effects”: they change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door). Though the HTTP standard makes it clear that POSTs are intended to always cause side-effects, and GET requests never to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself.
This is done as follows:

```python
>>> import urllib.request
>>> import urllib.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.parse.urlencode(data)
>>> print(url_values)  # The order may differ from below.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.urlopen(full_url)
```

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

**Headers**

We'll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites [1] dislike being browsed by programs, or send different versions to different browsers [2]. By default urllib identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/2.5`), which may confuse the site, or just plain not work. The way a browser identifies itself is through the `User-Agent` header [3]. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer [4].
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python'
headers = {'User-Agent': user_agent}

data = urllib.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urllib.request.urlopen(req) as response:
    the_page = response.read()

The response also has two useful methods. See the section on info and geturl which comes after we have a look at what happens when things go wrong.
Handling Exceptions

`urlopen` raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.

The exception classes are exported from the `urllib.error` module.

**URLError**

Often, `URLError` is raised because there is no network connection (no route to the specified server), or the specified server doesn’t exist. In this case, the exception raised will have a ‘reason’ attribute, which is a tuple containing an error code and a text error message.

e.g.

```python
>>> req = urllib.request.Request('http://www.pretend_server.org')
>>> try:
...     urllib.request.urlopen(req)
... except urllib.error.URLError as e:
...     print(e.reason)
... (4, 'getaddrinfo failed')
```

**HTTPError**

Every HTTP response from the server contains a numeric “status code”. Sometimes the status code indicates that the server is unable
to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a “redirection” that requests the client fetch the document from a different URL, urllib will handle that for you). For those it can’t handle, urlopen will raise an HTTPError. Typical errors include ‘404’ (page not found), ‘403’ (request forbidden), and ‘401’ (authentication required).

See section 10 of RFC 2616 for a reference on all the HTTP error codes.

The HTTPError instance raised will have an integer ‘code’ attribute, which corresponds to the error sent by the server.

Error Codes

Because the default handlers handle redirects (codes in the 300 range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

http.server.BaseHTTPRequestHandler.responses is a useful dictionary of response codes in that shows all the response codes used by RFC 2616. The dictionary is reproduced here for convenience

```python
# Table mapping response codes to messages; entries have the form {code: (shortmessage, longmessage)}.
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols', 'Switching to new protocol; obey Upgrade header'),

    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted', 'Request accepted, processing continues of
```
203: ('Non-Authoritative Information', 'Request fulfilled, nothing to add'),
204: ('No Content', 'Request fulfilled, nothing to add'),
205: ('Reset Content', 'Clear input form for further input.'),
206: ('Partial Content', 'Partial content follows.'),
300: ('Multiple Choices', 'Object has several resources -- see URI list'),
301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
302: ('Found', 'Object moved temporarily -- see URI list'),
303: ('See Other', 'Object moved -- see Method and URL list'),
304: ('Not Modified', 'Document has not changed since given time'),
305: ('Use Proxy', 'You must use proxy specified in Location resource.'),
307: ('Temporary Redirect', 'Object moved temporarily -- see URI list'),
400: ('Bad Request', 'Bad request syntax or unsupported method'),
401: ('Unauthorized', 'No permission -- see authorization schemes'),
402: ('Payment Required', 'No payment -- see charging schemes'),
403: ('Forbidden', 'Request forbidden -- authorization will not help'),
404: ('Not Found', 'Nothing matches the given URI'),
405: ('Method Not Allowed', 'Specified method is invalid for this server'),
406: ('Not Acceptable', 'URI not available in preferred format'),
407: ('Proxy Authentication Required', 'You must authenticate with this proxy before proceeding.'),
408: ('Request Timeout', 'Request timed out; try again later.'),
409: ('Conflict', 'Request conflict.'),
410: ('Gone', 'URI no longer exists and has been permanently removed.'),
411: ('Length Required', 'Client must specify Content-Length.'),
When an error is raised the server responds by returning an HTTP error code and an error page. You can use the HTTPError instance as a response on the page returned. This means that as well as the code attribute, it also has read, geturl, and info, methods as returned by the urllib.response module:

```python
>>> req = urllib.request.Request('http://www.python.org/fish.html')
>>> try:
...     urllib.request.urlopen(req)
... except urllib.error.HTTPError as e:
...     print(e.code)
...     print(e.read())
... 404
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Trans'
Wrapping it Up

So if you want to be prepared for HTTPError or URLError there are two basic approaches. I prefer the second approach.

Number 1

```python
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn’t fulfill the request.
    print('Error code: ', e.code)
except URLError as e:
    print('We failed to reach a server.
    print('Reason: ', e.reason)
else:
    # everything is fine
```

Note: The except HTTPError must come first, otherwise except URLError will also catch an HTTPError.

Number 2

```python
from urllib.request import Request, urlopen
```
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
    # everything is fine
info and geturl

The response returned by urlopen (or the HTTPError instance) has two useful methods info() and geturl() and is defined in the module urllib.response..

geturl - this returns the real URL of the page fetched. This is useful because urlopen (or the opener object used) may have followed a redirect. The URL of the page fetched may not be the same as the URL requested.

info - this returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an http.client.HTTPMessage instance.

Typical headers include ‘Content-length’, ‘Content-type’, and so on. See the Quick Reference to HTTP Headers for a useful listing of HTTP headers with brief explanations of their meaning and use.
Openers and Handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly-named `urllib.request.OpenerDirector`). Normally we have been using the default opener - via `urlopen` - but you can create custom openers. Openers use handlers. All the “heavy lifting” is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirections or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirections.

To create an opener, instantiate an `OpenerDirector`, and then call `.add_handler(some_handler_instance)` repeatedly.

Alternatively, you can use `build_opener`, which is a convenience function for creating opener objects with a single function call. `build_opener` adds several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to can handle proxies, authentication, and other common but slightly specialised situations.

`install_opener` can be used to make an `opener` object the (global) default opener. This means that calls to `urlopen` will use the opener you have installed.

Opener objects have an `open` method, which can be called directly to fetch urls in the same way as the `urlopen` function: there's no need to call `install_opener`, except as a convenience.
Basic Authentication

To illustrate creating and installing a handler we will use the HTTPBasicAuthHandler. For a more detailed discussion of this subject – including an explanation of how Basic Authentication works - see the Basic Authentication Tutorial.

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a ‘realm’. The header looks like: WWW-Authenticate: SCHEME realm="REALM".

e.g.

WWW-Authenticate: Basic realm="cPanel Users"

The client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is ‘basic authentication’. In order to simplify this process we can create an instance of HTTPBasicAuthHandler and an opener to use this handler.

The HTTPBasicAuthHandler uses an object called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a HTTPPasswordMgr. Frequently one doesn’t care what the realm is. In that case, it is convenient to use HTTPPasswordMgrWithDefaultRealm. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing None
as the realm argument to the `add_password` method.

The top-level URL is the first URL that requires authentication. URLs “deeper” than the URL you pass to `.add_password()` will also match.

```
# create a password manager
password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm

# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/
password_mgr.add_password(None, top_level_url, username)

handler = urllib.request.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib.request.build_opener(handler)

# use the opener to fetch a URL
opener.open(a_url)

# Install the opener.
# Now all calls to urllib.request.urlopen use our opener.
urllib.request.install_opener(opener)
```

**Note:** In the above example we only supplied our `HTTPBasicAuthHandler` to `build_opener`. By default openers have the handlers for normal situations – `ProxyHandler` (if a proxy setting such as an `http_proxy` environment variable is set), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `DataHandler`, `HTTPErrorProcessor`. 
top_level_url is in fact either a full URL (including the ‘http’ scheme component and the hostname and optionally the port number) e.g. "http://example.com/" or an “authority” (i.e. the hostname, optionally including the port number) e.g. "example.com" or "example.com:8080" (the latter example includes a port number). The authority, if present, must NOT contain the “userinfo” component - for example "joe:password@example.com" is not correct.
Proxies

`urllib` will auto-detect your proxy settings and use those. This is through the `ProxyHandler`, which is part of the normal handler chain when a proxy setting is detected. Normally that's a good thing, but there are occasions when it may not be helpful [5]. One way to do this is to setup our own `ProxyHandler`, with no proxies defined. This is done using similar steps to setting up a `Basic Authentication` handler:

```python
>>> proxy_support = urllib.request.ProxyHandler({})
>>> opener = urllib.request.build_opener(proxy_support)
>>> urllib.request.install_opener(opener)
```

**Note:** Currently `urllib.request` *does not* support fetching of `https` locations through a proxy. However, this can be enabled by extending `urllib.request` as shown in the recipe [6].

**Note:** `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.
Sockets and Layers

The Python support for fetching resources from the web is layered. urllib uses the `http.client` library, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages. By default the socket module has no timeout and can hang. Currently, the socket timeout is not exposed at the http.client or urllib.request levels. However, you can set the default timeout globally for all sockets using

```python
import socket
import urllib.request

# timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

# this call to urllib.request.urlopen now uses the default
# we have set in the socket module
req = urllib.request.Request('http://www.voidspace.org.uk')
response = urllib.request.urlopen(req)
```
Footnotes

This document was reviewed and revised by John Lee.


Browser sniffing is a very bad practice for website design - building sites using web standards is much more sensible. Unfortunately a lot of sites still send different versions to different browsers.

[2] The user agent for MSIE 6 is ‘Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)’

[3] For details of more HTTP request headers, see Quick Reference to HTTP Headers.

In my case I have to use a proxy to access the internet at work. If you attempt to fetch localhost URLs through this proxy it blocks them. IE is set to use the proxy, which urllib picks up on. In order to test scripts with a localhost server, I have to prevent urllib from using the proxy.

Argparse Tutorial

**author:** Tshepang Lekhonkhobe

This tutorial is intended to be a gentle introduction to **argparse**, the recommended command-line parsing module in the Python standard library.

**Note:** There are two other modules that fulfill the same task, namely **getopt** (an equivalent for getopt() from the C language) and the deprecated **optparse**. Note also that **argparse** is based on **optparse**, and therefore very similar in terms of usage.
Concepts

Let's show the sort of functionality that we are going to explore in this introductory tutorial by making use of the `ls` command:

```
$ ls
cpython devguide prog.py pypy rm-unused-function
$ ls -l
```

A few concepts we can learn from the four commands:

- The `ls` command is useful when run without any options at all. It defaults to displaying the contents of the current directory.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It's named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like `cp`, whose most basic usage is `cp SRC DEST`. The first position is what you want copied, and the second position is where you want it copied to.
• Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just showing the file names. The `-l` in that case is known as an optional argument.
• That's a snippet of the help text. It's very useful in that you can come across a program you have never used before, and can figure out how it works simply by reading its help text.
The basics

Let us start with a very simple example which does (almost) nothing:

```python
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Following is a result of running the code:

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]
optional arguments:
  -h, --help    show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Here is what is happening:

- Running the script without any options results in nothing displayed to stdout. Not so useful.
- The second one starts to display the usefulness of the `argparse` module. We have done almost nothing, but already we get a nice help message.
- The `--help` option, which can also be shortened to `-h`, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message, also for free.
Introducing Positional arguments

An example:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

And running the code:

```bash
$ python3 prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python3 prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help    show this help message and exit
$ python3 prog.py foo
foo
```

Here is what’s happening:

- We’ve added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I’ve named it `echo` so that it’s in line with its function.

- Calling our program now requires us to specify an option.

- The `parse_args()` method actually returns some data from the
options specified, in this case, `echo`.
- The variable is some form of ‘magic’ that `argparse` performs for free (i.e. no need to specify which variable that value is stored in). You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don’t know what it does, other than by guessing or by reading the source code. So, let’s make it a bit more useful:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

And we get:

```
$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo            echo the string you use here

optional arguments:
  -h, --help      show this help message and exit
```

Now, how about doing something even more useful:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square")
```
```python
args = parser.parse_args()
print(args.square**2)
```

Following is a result of running the code:

```
$ python3 prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

That didn't go so well. That's because `argparse` treats the options we give it as strings, unless we tell it otherwise. So, let's tell `argparse` to treat that input as an integer:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number", type=int)
args = parser.parse_args()
print(args.square**2)
```

Following is a result of running the code:

```
$ python3 prog.py 4
16
$ python3 prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value:
```

That went well. The program now even helpfully quits on bad illegal input before proceeding.
Introducing Optional arguments

So far we have been playing with positional arguments. Let us have a look on how to add optional ones:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

And the output:

```
$ python3 prog.py --verbosity 1
verbosity turned on
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]
optional arguments:
   -h, --help               show this help message and exit
   --verbosity VERBOSITY    increase output verbosity
$ python3 prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Here is what is happening:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- To show that the option is actually optional, there is no error when
running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case args.verbosity, is given None as a value, which is the reason it fails the truth test of the if statement.

- The help message is a bit different.
- When using the --verbosity option, one must also specify some value, any value.

The above example accepts arbitrary integer values for --verbosity, but for our simple program, only two values are actually useful, True or False. Let’s modify the code accordingly:

```python
import argparse
def main()
    parser = argparse.ArgumentParser()
    parser.add_argument("--verbose", help="increase output verbosity", action="store_true")
    args = parser.parse_args()
    if args.verbose:
        print("verbosity turned on")
```

And the output:

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
    -h, --help    show this help message and exit
    --verbose     increase output verbosity
```
Here is what is happening:

- The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value "store_true". This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

**Short options**

If you are familiar with command line usage, you will notice that I haven’t yet touched on the topic of short versions of the options. It’s quite simple:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

And here goes:

```bash
$ python3 prog.py -v
verbosity turned on
$ python3 prog.py --help
usage: prog.py [-h] [-v]
optional arguments:
```
-h, --help show this help message and exit
-v, --verbose increase output verbosity

Note that the new ability is also reflected in the help text.
Combining Positional and Optional arguments

Our program keeps growing in complexity:

```python
import argparse
parser = argparse.ArgumentParser(
    prog="square", type=int,
    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print("the square of {} equals {}".format(args.square))
else:
    print(answer)
```

And now the output:

```
$ python3 prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required
$ python3 prog.py 4
16
$ python3 prog.py 4 --verbose
the square of 4 equals 16
$ python3 prog.py --verbose 4
the square of 4 equals 16
```

- We’ve brought back a positional argument, hence the complaint.
- Note that the order does not matter.
How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
parser.add_argument("square")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

And the output:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python3 prog.py 4 -v 1
4^2 == 16
$ python3 prog.py 4 -v 2
the square of 4 equals 16
$ python3 prog.py 4 -v 3
16
```

These all look good except the last one, which exposes a bug in our program. Let’s fix it by restricting the values the `--verbosity` option...
can accept:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
    help="display a square of a given")
parser.add_argument("-v", "--verbosity", type=int, choices=int
    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

And the output:

```
$ python3 prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                     display a square of a given

optional arguments:
  -h, --help                 show this help message and exit
  -v {0,1,2}, --verbosity {0,1,2} increase output verbosity
```

Note that the change also reflects both in the error message as well as the help string.
Now, let's use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of `python --help`):

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int, help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count", help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

We have introduced another action, “count”, to count the number of occurrences of a specific optional arguments:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
4^2 == 16
$ python3 prog.py 4 -vv
the square of 4 equals 16
$ python3 prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python3 prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python3 prog.py 4 -h
```
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity

$ python3 prog.py 4 -vvv
16

- Yes, it's now more of a flag (similar to action="store_true") in the previous version of our script. That should explain the complaint.
- It also behaves similar to “store_true” action.
- Now here's a demonstration of what the “count” action gives. You've probably seen this sort of usage before.
- And if you don't specify the -v flag, that flag is considered to have None value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn't very informative on the new ability our script has acquired, but that can always be fixed by improving the documentation for our script (e.g. via the help keyword argument).
- That last output exposes a bug in our program.

Let's fix:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
```
```python
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

And this is what it gives:

```bash
$ python3 prog.py 4 -vv
the square of 4 equals 16
$ python3 prog.py 4 -vvvv
the square of 4 equals 16
$ python3 prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '==' not supported between instances of 'NoneType' and 'int'
```

- First output went well, and fixes the bug we had before. That is, we want any value \( \geq 2 \) to be as verbose as possible.
- Third output not so good.

Let's fix that bug:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
p reassuring help="increase output verbosity")
```
Here's the code snippet and its explanation:

```python
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

We've just introduced yet another keyword, `default`. We've set it to `0` in order to make it comparable to the other `int` values. Remember that by default, if an optional argument isn't specified, it gets the `None` value, and that cannot be compared to an `int` value (hence the `TypeError` exception).

And:

```
$ python3 prog.py 4
16
```

You can go quite far just with what we've learned so far, and we have only scratched the surface. The `argparse` module is very powerful, and we'll explore a bit more of it before we end this tutorial.
Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count")
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
elif args.verbosity >= 1:
    print("{}^{} == {}".format(args.x, args.y, answer))
else:
    print(answer)
```

Output:

```
$ python3 prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required
$ python3 prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x          the base
  y          the exponent

optional arguments:
  -h, --help  show this help message and exit
  -v, --verbosity
```
Notice that so far we’ve been using verbosity level to *change* the text that gets displayed. The following example instead uses verbosity level to display *more* text instead:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count")
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print("Running '{}".format(__file__))
if args.verbosity >= 1:
    print("{}^{} == ".format(args.x, args.y), end="")
print(answer)
```

Output:

```
$ python3 prog.py 4 2
16
$ python3 prog.py 4 2 -v
4^2 == 16
$ python3 prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

**Conflicting options**

So far, we have been working with two methods of an
argparse.ArgumentParser instance. Let's introduce a third one, add_mutually_exclusive_group(). It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense: we'll introduce the **quiet** option, which will be the opposite of the **verbose** one:

```python
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group() 
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

Our program is now simpler, and we've lost some functionality for the sake of demonstration. Anyways, here's the output:

```
$ python3 prog.py 4 2
4^2 == 16
$ python3 prog.py 4 2 -q
16
$ python3 prog.py 4 2 -v
4 to the power 2 equals 16
```
That should be easy to follow. I’ve added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don’t know:

```python
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")

args = parser.parse_args()

answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}").format(args.x, args.y, answer)
else:
    print("{}^{} == {}").format(args.x, args.y, answer)
```

Note that slight difference in the usage text. Note the `[-v | -q]`, which tells us that we can either use `-v` or `-q`, but not both at the
same time:

```
$ python3 prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x          the base
  y          the exponent

optional arguments:
  -h, --help  show this help message and exit
  -v, --verbose
  -q, --quiet
```
Conclusion

The argparse module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.
An introduction to the ipaddress module

Overview

This document aims to provide a gentle introduction to the ipaddress module. It is aimed primarily at users that aren't already familiar with IP networking terminology, but may also be useful to network engineers wanting an overview of how ipaddress represents IP network addressing concepts.
Creating Address/Network/Interface objects

Since `ipaddress` is a module for inspecting and manipulating IP addresses, the first thing you'll want to do is create some objects. You can use `ipaddress` to create objects from strings and integers.

A Note on IP Versions

For readers that aren't particularly familiar with IP addressing, it's important to know that the Internet Protocol is currently in the process of moving from version 4 of the protocol to version 6. This transition is occurring largely because version 4 of the protocol doesn't provide enough addresses to handle the needs of the whole world, especially given the increasing number of devices with direct connections to the internet.

Explaining the details of the differences between the two versions of the protocol is beyond the scope of this introduction, but readers need to at least be aware that these two versions exist, and it will sometimes be necessary to force the use of one version or the other.

IP Host Addresses

Addresses, often referred to as “host addresses” are the most basic unit when working with IP addressing. The simplest way to create addresses is to use the `ipaddress.ip_address()` factory function, which automatically determines whether to create an IPv4 or IPv6 address based on the passed in value:
Addresses can also be created directly from integers. Values that will fit within 32 bits are assumed to be IPv4 addresses:

```python
>>> ipaddress.ip_address('192.0.2.1')
IPv4Address('192.0.2.1')
>>> ipaddress.ip_address('2001:DB8::1')
IPv6Address('2001:db8::1')
```

To force the use of IPv4 or IPv6 addresses, the relevant classes can be invoked directly. This is particularly useful to force creation of IPv6 addresses for small integers:

```python
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.ip_address(42540766412825928569039849)
IPv6Address('2001:db8::1')
```

**Defining Networks**

Host addresses are usually grouped together into IP networks, so `ipaddress` provides a way to create, inspect and manipulate network definitions. IP network objects are constructed from strings that define the range of host addresses that are part of that network. The simplest form for that information is a “network address/network prefix” pair, where the prefix defines the number of leading bits that are compared to determine whether or not an address is part of the network and the
network address defines the expected value of those bits.

As for addresses, a factory function is provided that determines the correct IP version automatically:

```python
>>> ipaddress.ip_network('192.0.2.0/24')
IPv4Network('192.0.2.0/24')
>>> ipaddress.ip_network('2001:db8::0/96')
IPv6Network('2001:db8::/96')
```

Network objects cannot have any host bits set. The practical effect of this is that `192.0.2.1/24` does not describe a network. Such definitions are referred to as interface objects since the ip-on-a-network notation is commonly used to describe network interfaces of a computer on a given network and are described further in the next section.

By default, attempting to create a network object with host bits set will result in `ValueError` being raised. To request that the additional bits instead be coerced to zero, the flag `strict=False` can be passed to the constructor:

```python
>>> ipaddress.ip_network('192.0.2.1/24')
Traceback (most recent call last):
 ...  
ValueError: 192.0.2.1/24 has host bits set
>>> ipaddress.ip_network('192.0.2.1/24', strict=False)
IPv4Network('192.0.2.0/24')
```

While the string form offers significantly more flexibility, networks can also be defined with integers, just like host addresses. In this case, the network is considered to contain only the single address identified by the integer, so the network prefix includes the entire network address:
As with addresses, creation of a particular kind of network can be forced by calling the class constructor directly instead of using the factory function.

**Host Interfaces**

As mentioned just above, if you need to describe an address on a particular network, neither the address nor the network classes are sufficient. Notation like `192.0.2.1/24` is commonly used by network engineers and the people who write tools for firewalls and routers as shorthand for “the host 192.0.2.1 on the network 192.0.2.0/24”.

Accordingly, `ipaddress` provides a set of hybrid classes that associate an address with a particular network. The interface for creation is identical to that for defining network objects, except that the address portion isn’t constrained to being a network address.

```python
>>> ipaddress.ip_network(3221225984)
IPv4Network('192.0.2.0/32')
>>> ipaddress.ip_network(42540764112825928569039849)
IPv6Network('2001:db8::/128')
```

```python
>>> ipaddress.ip_interface('192.0.2.1/24')
IPv4Interface('192.0.2.1/24')
>>> ipaddress.ip_interface('2001:db8::1/96')
IPv6Interface('2001:db8::1/96')
```

Integer inputs are accepted (as with networks), and use of a particular IP version can be forced by calling the relevant constructor directly.
Inspecting Address/Network/Interface Objects

You’ve gone to the trouble of creating an IPv(4|6) (Address|Network|Interface) object, so you probably want to get information about it. `ipaddress` tries to make doing this easy and intuitive.

Extracting the IP version:

```python
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr6 = ipaddress.ip_address('2001:db8::1')
>>> addr6.version
6
>>> addr4.version
4
```

Obtaining the network from an interface:

```python
>>> host4 = ipaddress.ip_interface('192.0.2.1/24')
>>> host4.network
IPv4Network('192.0.2.0/24')
>>> host6 = ipaddress.ip_interface('2001:db8::1/96')
>>> host6.network
IPv6Network('2001:db8::/96')
```

Finding out how many individual addresses are in a network:

```python
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.num_addresses
256
>>> net6 = ipaddress.ip_network('2001:db8::/96')
>>> net6.num_addresses
```
Iterating through the “usable” addresses on a network:

```python
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> for x in net4.hosts():
...   print(x)
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
...
192.0.2.252
192.0.2.253
192.0.2.254
```

Obtaining the netmask (i.e. set bits corresponding to the network prefix) or the hostmask (any bits that are not part of the netmask):

```python
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.netmask
IPv4Address('255.255.255.0')
>>> net4.hostmask
IPv4Address('0.0.0.255')

>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6.netmask
>>> net6.hostmask
IPv6Address('::ffff:ffff')
```

Exploding or compressing the address:

```python
>>> addr6.exploded
'2001:0db8:0000:0000:0000:0000:0000:0001'
>>> addr6.compressed
'2001:db8::1'
```
While IPv4 doesn't support explosion or compression, the associated objects still provide the relevant properties so that version neutral code can easily ensure the most concise or most verbose form is used for IPv6 addresses while still correctly handling IPv4 addresses.
Networks as lists of Addresses

It’s sometimes useful to treat networks as lists. This means it is possible to index them like this:

```python
>>> net4[1]
IPv4Address('192.0.2.1')
>>> net4[-1]
IPv4Address('192.0.2.255')
>>> net6[1]
IPv6Address('2001:db8::1')
>>> net6[-1]
IPv6Address('2001:db8::ffff:ffff')
```

It also means that network objects lend themselves to using the list membership test syntax like this:

```python
if address in network:
    # do something
```

Containment testing is done efficiently based on the network prefix:

```python
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr4 in ipaddress.ip_network('192.0.2.0/24')
True
>>> addr4 in ipaddress.ip_network('192.0.3.0/24')
False
```
Comparisons

`ipaddress` provides some simple, hopefully intuitive ways to compare objects, where it makes sense:

```python
>>> ipaddress.ip_address('192.0.2.1') < ipaddress.ip_address('192.0.2.2')
True
```

A `TypeError` exception is raised if you try to compare objects of different versions or different types.
Using IP Addresses with other modules

Other modules that use IP addresses (such as `socket`) usually won’t accept objects from this module directly. Instead, they must be coerced to an integer or string that the other module will accept:

```python
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> str(addr4)
'192.0.2.1'
>>> int(addr4)
3221225985
```
Getting more detail when instance creation fails

When creating address/network/interface objects using the version-agnostic factory functions, any errors will be reported as `ValueError` with a generic error message that simply says the passed in value was not recognized as an object of that type. The lack of a specific error is because it’s necessary to know whether the value is `supposed` to be IPv4 or IPv6 in order to provide more detail on why it has been rejected.

To support use cases where it is useful to have access to this additional detail, the individual class constructors actually raise the `ValueError` subclasses `ipaddress.AddressValueError` and `ipaddress.NetmaskValueError` to indicate exactly which part of the definition failed to parse correctly.

The error messages are significantly more detailed when using the class constructors directly. For example:

```python
>>> ipaddress.ip_address("192.168.0.256")
Traceback (most recent call last):
  ...  
ValueError: '192.168.0.256' does not appear to be an IPv4 or IPv6 address

>>> ipaddress.IPv4Address("192.168.0.256")
Traceback (most recent call last):
  ...  
ipaddress.AddressValueError: Octet 256 (> 255) not permitted in '192.168.0.256'

>>> ipaddress.ip_network("192.168.0.1/64")
Traceback (most recent call last):
  ...  
ValueError: '192.168.0.1/64' does not appear to be a
```
>>> ipaddress.IPv4Network("192.168.0.1/64")
Traceback (most recent call last):
...
  ipaddress.NetmaskValueError: '64' is not a valid net

However, both of the module specific exceptions have `ValueError` as their parent class, so if you’re not concerned with the particular type of error, you can still write code like the following:

```python
try:
    network = ipaddress.IPv4Network(address)
except ValueError:
    print('address/netmask is invalid for IPv4:', address)
```
Argument Clinic How-To

**author:** Larry Hastings

**Abstract**

Argument Clinic is a preprocessor for CPython C files. Its purpose is to automate all the boilerplate involved with writing argument parsing code for “builtins”. This document shows you how to convert your first C function to work with Argument Clinic, and then introduces some advanced topics on Argument Clinic usage.

Currently Argument Clinic is considered internal-only for CPython. Its use is not supported for files outside CPython, and no guarantees are made regarding backwards compatibility for future versions. In other words: if you maintain an external C extension for CPython, you're welcome to experiment with Argument Clinic in your own code. But the version of Argument Clinic that ships with the next version of CPython could be totally incompatible and break all your code.
The Goals Of Argument Clinic

Argument Clinic’s primary goal is to take over responsibility for all argument parsing code inside CPython. This means that, when you convert a function to work with Argument Clinic, that function should no longer do any of its own argument parsing—the code generated by Argument Clinic should be a “black box” to you, where CPython calls in at the top, and your code gets called at the bottom, with `PyObject *args` (and maybe `PyObject *kwargs`) magically converted into the C variables and types you need.

In order for Argument Clinic to accomplish its primary goal, it must be easy to use. Currently, working with CPython’s argument parsing library is a chore, requiring maintaining redundant information in a surprising number of places. When you use Argument Clinic, you don’t have to repeat yourself.

Obviously, no one would want to use Argument Clinic unless it’s solving their problem—and without creating new problems of its own. So it’s paramount that Argument Clinic generate correct code. It’d be nice if the code was faster, too, but at the very least it should not introduce a major speed regression. (Eventually Argument Clinic should make a major speedup possible—we could rewrite its code generator to produce tailor-made argument parsing code, rather than calling the general-purpose CPython argument parsing library. That would make for the fastest argument parsing possible!)

Additionally, Argument Clinic must be flexible enough to work with any approach to argument parsing. Python has some functions with some very strange parsing behaviors; Argument Clinic’s goal is to support all of them.
Finally, the original motivation for Argument Clinic was to provide introspection “signatures” for CPython builtins. It used to be, the introspection query functions would throw an exception if you passed in a builtin. With Argument Clinic, that’s a thing of the past!

One idea you should keep in mind, as you work with Argument Clinic: the more information you give it, the better job it’ll be able to do. Argument Clinic is admittedly relatively simple right now. But as it evolves it will get more sophisticated, and it should be able to do many interesting and smart things with all the information you give it.
Basic Concepts And Usage

Argument Clinic ships with CPython; you'll find it in Tools/clinic/clinic.py. If you run that script, specifying a C file as an argument:

```bash
$ python3 Tools/clinic/clinic.py foo.c
```

Argument Clinic will scan over the file looking for lines that look exactly like this:

```
/*[clinic input]
```

When it finds one, it reads everything up to a line that looks exactly like this:

```
[clinic start generated code]*/
```

Everything in between these two lines is input for Argument Clinic. All of these lines, including the beginning and ending comment lines, are collectively called an Argument Clinic “block”.

When Argument Clinic parses one of these blocks, it generates output. This output is rewritten into the C file immediately after the block, followed by a comment containing a checksum. The Argument Clinic block now looks like this:

```
/*[clinic input]
... clinic input goes here ...
[clinic start generated code]*/
... clinic output goes here ...
/*[clinic end generated code: checksum=...]*/
```
If you run Argument Clinic on the same file a second time, Argument Clinic will discard the old output and write out the new output with a fresh checksum line. However, if the input hasn’t changed, the output won’t change either.

You should never modify the output portion of an Argument Clinic block. Instead, change the input until it produces the output you want. (That’s the purpose of the checksum—to detect if someone changed the output, as these edits would be lost the next time Argument Clinic writes out fresh output.)

For the sake of clarity, here’s the terminology we’ll use with Argument Clinic:

- The first line of the comment (/*[clinic input]*) is the start line.
- The last line of the initial comment ([clinic start generated code]*) is the end line.
- The last line (/*[clinic end generated code: checksum=...]*/) is the checksum line.
- In between the start line and the end line is the input.
- In between the end line and the checksum line is the output.
- All the text collectively, from the start line to the checksum line inclusively, is the block. (A block that hasn’t been successfully processed by Argument Clinic yet doesn’t have output or a checksum line, but it’s still considered a block.)
Converting Your First Function

The best way to get a sense of how Argument Clinic works is to convert a function to work with it. Here, then, are the bare minimum steps you’d need to follow to convert a function to work with Argument Clinic. Note that for code you plan to check in to CPython, you really should take the conversion farther, using some of the advanced concepts you'll see later on in the document (like “return converters” and “self converters”). But we’ll keep it simple for this walkthrough so you can learn.

Let’s dive in!

0. Make sure you’re working with a freshly updated checkout of the CPython trunk.

1. Find a Python built-in that calls either `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()`, and hasn’t been converted to work with Argument Clinic yet. For my example I’m using `_pickle.Pickler.dump()`.

2. If the call to the `PyArg_Parse` function uses any of the following format units:

   
   | 0& |
   | 0! |
   | es |
   | es# |
   | et |
   | et# |

   or if it has multiple calls to `PyArg_ParseTuple()`, you should choose a different function. Argument Clinic does support all of
these scenarios. But these are advanced topics—let's do something simpler for your first function.

Also, if the function has multiple calls to `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()` where it supports different types for the same argument, or if the function uses something besides `PyArg_Parse` functions to parse its arguments, it probably isn't suitable for conversion to Argument Clinic. Argument Clinic doesn't support generic functions or polymorphic parameters.

3. Add the following boilerplate above the function, creating our block:

```c
/*[clinic input]
[clinic start generated code]*/
```

4. Cut the docstring and paste it in between the `[clinic]` lines, removing all the junk that makes it a properly quoted C string. When you're done you should have just the text, based at the left margin, with no line wider than 80 characters. (Argument Clinic will preserve indents inside the docstring.)

If the old docstring had a first line that looked like a function signature, throw that line away. (The docstring doesn’t need it anymore—when you use `help()` on your builtin in the future, the first line will be built automatically based on the function's signature.)

Sample:

```c
/*[clinic input]
Write a pickled representation of obj to the open
```
5. If your docstring doesn’t have a “summary” line, Argument Clinic will complain. So let’s make sure it has one. The “summary” line should be a paragraph consisting of a single 80-column line at the beginning of the docstring.

(Our example docstring consists solely of a summary line, so the sample code doesn’t have to change for this step.)

6. Above the docstring, enter the name of the function, followed by a blank line. This should be the Python name of the function, and should be the full dotted path to the function—it should start with the name of the module, include any sub-modules, and if the function is a method on a class it should include the class name too.

Sample:

/*[clinic input] _pickle.Pickler.dump
Write a pickled representation of obj to the open file.[clinic start generated code]*/

7. If this is the first time that module or class has been used with Argument Clinic in this C file, you must declare the module and/or class. Proper Argument Clinic hygiene prefers declaring these in a separate block somewhere near the top of the C file, in the same way that include files and statics go at the top. (In our sample code we’ll just show the two blocks next to each other.)

The name of the class and module should be the same as the one
seen by Python. Check the name defined in the `PyModuleDef` or `PyTypeObject` as appropriate.

When you declare a class, you must also specify two aspects of its type in C: the type declaration you’d use for a pointer to an instance of this class, and a pointer to the `PyTypeObject` for this class.

Sample:

```c
/*[clinic input]*/
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
/*[clinic start generated code]*/

/*[clinic input]*/
_pickle.Pickler.dump

Write a pickled representation of obj to the open
/*[clinic start generated code]*/
```

8. Declare each of the parameters to the function. Each parameter should get its own line. All the parameter lines should be indented from the function name and the docstring.

The general form of these parameter lines is as follows:

```
name_of_parameter: converter
```

If the parameter has a default value, add that after the converter:

```
name_of_parameter: converter = default_value
```

Argument Clinic’s support for “default values” is quite
sophisticated; please see the section below on default values for more information.

Add a blank line below the parameters.

What’s a “converter”? It establishes both the type of the variable used in C, and the method to convert the Python value into a C value at runtime. For now you’re going to use what’s called a “legacy converter”—a convenience syntax intended to make porting old code into Argument Clinic easier.

For each parameter, copy the “format unit” for that parameter from the PyArg_Parse() format argument and specify that as its converter, as a quoted string. (“format unit” is the formal name for the one-to-three character substring of the format parameter that tells the argument parsing function what the type of the variable is and how to convert it. For more on format units please see Parsing arguments and building values.)

For multicharacter format units like z#, use the entire two-or-three character string.

Sample:

```c
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'

Write a pickled representation of obj to the open
```
9. If your function has `|` in the format string, meaning some parameters have default values, you can ignore it. Argument Clinic infers which parameters are optional based on whether or not they have default values.

If your function has `$` in the format string, meaning it takes keyword-only arguments, specify `*` on a line by itself before the first keyword-only argument, indented the same as the parameter lines.

(`_pickle.Pickler.dump` has neither, so our sample is unchanged.)

10. If the existing C function calls `PyArg_ParseTuple()` (as opposed to `PyArg_ParseTupleAndKeywords()`), then all its arguments are positional-only.

To mark all parameters as positional-only in Argument Clinic, add a `/` on a line by itself after the last parameter, indented the same as the parameter lines.

Currently this is all-or-nothing; either all parameters are positional-only, or none of them are. (In the future Argument Clinic may relax this restriction.)

Sample:

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler"
[clinic start generated code]*/
It's helpful to write a per-parameter docstring for each parameter. But per-parameter docstrings are optional; you can skip this step if you prefer.

Here's how to add a per-parameter docstring. The first line of the per-parameter docstring must be indented further than the parameter definition. The left margin of this first line establishes the left margin for the whole per-parameter docstring; all the text you write will be indented by this amount. You can write as much text as you like, across multiple lines if you wish.

Sample:

```python
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
/

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```
12. Save and close the file, then run `Tools/clinic/clinic.py` on it. With luck everything worked—your block now has output, and a `.c.h` file has been generated! Reopen the file in your text editor to see:

```c
/*[clinic input]
 _pickle.Pickler.dump

   obj: 'O'
   The object to be pickled.
/

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

static PyObject *
_picklle_Pickler_dump(PicklerObject *self, PyObject *obj)

/*[clinic end generated code: output=87ecad1261e02ac7 input=552eb1c0f52260d9]*/
```

Obviously, if Argument Clinic didn’t produce any output, it’s because it found an error in your input. Keep fixing your errors and retrying until Argument Clinic processes your file without complaint.

For readability, most of the glue code has been generated to a `.c.h` file. You’ll need to include that in your original `.c` file, typically right after the clinic module block:

```
#include "clinic/_pickle.c.h"
```
13. Double-check that the argument-parsing code Argument Clinic generated looks basically the same as the existing code.

First, ensure both places use the same argument-parsing function. The existing code must call either PyArg_ParseTuple() or PyArg_ParseTupleAndKeywords(); ensure that the code generated by Argument Clinic calls the exact same function.

Second, the format string passed in to PyArg_ParseTuple() or PyArg_ParseTupleAndKeywords() should be exactly the same as the hand-written one in the existing function, up to the colon or semi-colon.

(Argument Clinic always generates its format strings with a : followed by the name of the function. If the existing code’s format string ends with ;, to provide usage help, this change is harmless —don’t worry about it.)

Third, for parameters whose format units require two arguments (like a length variable, or an encoding string, or a pointer to a conversion function), ensure that the second argument is exactly the same between the two invocations.

Fourth, inside the output portion of the block you’ll find a preprocessor macro defining the appropriate static PyMethodDef structure for this builtin:

```
#define __PICKLE_PICKLER_DUMP_METHODDEF  \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O,  \
"pickle dump", 0}
```

This static structure should be exactly the same as the existing
static PyMethodDef structure for this builtin.

If any of these items differ in any way, adjust your Argument Clinic function specification and rerun Tools/clinic/clinic.py until they are the same.

14. Notice that the last line of its output is the declaration of your “impl” function. This is where the builtin’s implementation goes. Delete the existing prototype of the function you’re modifying, but leave the opening curly brace. Now delete its argument parsing code and the declarations of all the variables it dumps the arguments into. Notice how the Python arguments are now arguments to this impl function; if the implementation used different names for these variables, fix it.

Let’s reiterate, just because it’s kind of weird. Your code should now look like this:

```c
static return_type
your_function_impl(...) /*[clinic end generated code: checksum=...]*/
{
...
```

Argument Clinic generated the checksum line and the function prototype just above it. You should write the opening (and closing) curly braces for the function, and the implementation inside.

Sample:

```c
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject "&Pickler
[clinic start generated code]*/
```
_pickle.Pickler.dump

obj: 'O'
The object to be pickled.
/

Write a pickled representation of obj to the open file.

PyDoc_STRVAR(__pickle_Pickler_dump__doc__,
"Write a pickled representation of obj to the open file.
"
...}

static PyObject * _pickle_Pickler_dump_impl(PicklerObject *self, Py

/*[clinic start generated code]*/{  
  /* Check whether the Pickler was initialized
     Developers often forget to call __init__().
     This would trigger a segfault without this check
  */
  if (self->write == NULL) {  
    PyErr_Format(PicklingError,
      "Pickler.__init__() was not called
    Py_TYPE(self)->tp_name);
    return NULL;
  }

  if (_Pickler_ClearBuffer(self) < 0)  
    return NULL;

  ...

15. Remember the macro with the PyMethodDef structure for this
function? Find the existing `PyMethodDef` structure for this function and replace it with a reference to the macro. (If the builtin is at module scope, this will probably be very near the end of the file; if the builtin is a class method, this will probably be below but relatively near to the implementation.)

Note that the body of the macro contains a trailing comma. So when you replace the existing static `PyMethodDef` structure with the macro, *don’t* add a comma to the end.

Sample:

```c
static struct PyMethodDef Pickler_methods[] = {
    __PICKLE_PICKLER_DUMP_METHODDEF
    __PICKLE_PICKLER_CLEAR_MEMO_METHODDEF
    {NULL, NULL} /* sentinel */
};
```

16. Compile, then run the relevant portions of the regression-test suite. This change should not introduce any new compile-time warnings or errors, and there should be no externally-visible change to Python’s behavior.

Well, except for one difference: `inspect.signature()` run on your function should now provide a valid signature!

Congratulations, you’ve ported your first function to work with Argument Clinic!
Advanced Topics

Now that you've had some experience working with Argument Clinic, it's time for some advanced topics.

Symbolic default values

The default value you provide for a parameter can't be any arbitrary expression. Currently the following are explicitly supported:

- Numeric constants (integer and float)
- String constants
- True, False, and None
- Simple symbolic constants like `sys.maxsize`, which must start with the name of the module

In case you're curious, this is implemented in `from_builtins()` in `Lib/inspect.py`.

(In the future, this may need to get even more elaborate, to allow full expressions like `CONSTANT - 1`.)

Renaming the C functions and variables generated by Argument Clinic

Argument Clinic automatically names the functions it generates for you. Occasionally this may cause a problem, if the generated name collides with the name of an existing C function. There's an easy solution: override the names used for the C functions. Just add the keyword "as" to your function declaration line, followed by the function name you wish to use. Argument Clinic will use that function...
name for the base (generated) function, then add "_impl" to the end and use that for the name of the impl function.

For example, if we wanted to rename the C function names generated for `pickle.Pickler.dump`, it'd look like this:

```c
/*[clinic input]
pickle.Pickler.dump as pickler_dumper
...
```

The base function would now be named `pickler_dumper()`, and the impl function would now be named `pickler_dumper_impl()`.

Similarly, you may have a problem where you want to give a parameter a specific Python name, but that name may be inconvenient in C. Argument Clinic allows you to give a parameter different names in Python and in C, using the same "as" syntax:

```c
/*[clinic input]
pickle.Pickler.dump

    obj: object
    file as file_obj: object
    protocol: object = NULL
    *
    fix_imports: bool = True
```

Here, the name used in Python (in the signature and the `keywords` array) would be `file`, but the C variable would be named `file_obj`.

You can use this to rename the `self` parameter too!

**Converting functions using PyArg_UnpackTuple**
To convert a function parsing its arguments with `PyArg_UnpackTuple()`, simply write out all the arguments, specifying each as an object. You may specify the type argument to cast the type as appropriate. All arguments should be marked positional-only (add a / on a line by itself after the last argument).

Currently the generated code will use `PyArg_ParseTuple()`, but this will change soon.

Optional Groups

Some legacy functions have a tricky approach to parsing their arguments: they count the number of positional arguments, then use a switch statement to call one of several different `PyArg_ParseTuple()` calls depending on how many positional arguments there are. (These functions cannot accept keyword-only arguments.) This approach was used to simulate optional arguments back before `PyArg_ParseTupleAndKeywords()` was created.

While functions using this approach can often be converted to use `PyArg_ParseTupleAndKeywords()`, optional arguments, and default values, it’s not always possible. Some of these legacy functions have behaviors `PyArg_ParseTupleAndKeywords()` doesn’t directly support. The most obvious example is the built-in function `range()`, which has an optional argument on the left side of its required argument! Another example is `curses.window.addch()`, which has a group of two arguments that must always be specified together. (The arguments are called x and y; if you call the function passing in x, you must also pass in y—and if you don’t pass in x you may not pass in y either.)
In any case, the goal of Argument Clinic is to support argument parsing for all existing CPython builtins without changing their semantics. Therefore Argument Clinic supports this alternate approach to parsing, using what are called *optional groups*. Optional groups are groups of arguments that must all be passed in together. They can be to the left or the right of the required arguments. They can *only* be used with positional-only parameters.

**Note:** Optional groups are *only* intended for use when converting functions that make multiple calls to `PyArg_ParseTuple()`! Functions that use *any* other approach for parsing arguments should *almost never* be converted to Argument Clinic using optional groups. Functions using optional groups currently cannot have accurate signatures in Python, because Python just doesn’t understand the concept. Please avoid using optional groups wherever possible.

To specify an optional group, add a `[` on a line by itself before the parameters you wish to group together, and a `]` on a line by itself after these parameters. As an example, here’s how `curses.window.addch` uses optional groups to make the first two parameters and the last parameter optional:

```c
/*[clinic input]
curses.window.addch

[  
  x: int
  X-coordinate.
  y: int
  Y-coordinate.
]```
ch: object
Charater to add.

[
  attr: long
  Attributes for the character.
]
/

Notes:

- For every optional group, one additional parameter will be passed into the impl function representing the group. The parameter will be an int named `group_{direction}_{number}`, where `{direction}` is either right or left depending on whether the group is before or after the required parameters, and `{number}` is a monotonically increasing number (starting at 1) indicating how far away the group is from the required parameters. When the impl is called, this parameter will be set to zero if this group was unused, and set to non-zero if this group was used. (By used or unused, I mean whether or not the parameters received arguments in this invocation.)
- If there are no required arguments, the optional groups will behave as if they're to the right of the required arguments.
- In the case of ambiguity, the argument parsing code favors parameters on the left (before the required parameters).
- Optional groups can only contain positional-only parameters.
- Optional groups are only intended for legacy code. Please do not use optional groups for new code.

Using real Argument Clinic converters, instead of “legacy converters”
To save time, and to minimize how much you need to learn to achieve your first port to Argument Clinic, the walkthrough above tells you to use “legacy converters”. “Legacy converters” are a convenience, designed explicitly to make porting existing code to Argument Clinic easier. And to be clear, their use is acceptable when porting code for Python 3.4.

However, in the long term we probably want all our blocks to use Argument Clinic's real syntax for converters. Why? A couple reasons:

- The proper converters are far easier to read and clearer in their intent.
- There are some format units that are unsupported as “legacy converters”, because they require arguments, and the legacy converter syntax doesn't support specifying arguments.
- In the future we may have a new argument parsing library that isn't restricted to what `PyArg_ParseTuple()` supports; this flexibility won't be available to parameters using legacy converters.

Therefore, if you don't mind a little extra effort, please use the normal converters instead of legacy converters.

In a nutshell, the syntax for Argument Clinic (non-legacy) converters looks like a Python function call. However, if there are no explicit arguments to the function (all functions take their default values), you may omit the parentheses. Thus `bool` and `bool()` are exactly the same converters.

All arguments to Argument Clinic converters are keyword-only. All Argument Clinic converters accept the following arguments:

```
c_default
```

The default value for this parameter when defined in C.
Specifically, this will be the initializer for the variable declared in the “parse function”. See the section on default values for how to use this. Specified as a string.

**annotation**

The annotation value for this parameter. Not currently supported, because PEP 8 mandates that the Python library may not use annotations.

In addition, some converters accept additional arguments. Here is a list of these arguments, along with their meanings:

**accept**

A set of Python types (and possibly pseudo-types); this restricts the allowable Python argument to values of these types. (This is not a general-purpose facility; as a rule it only supports specific lists of types as shown in the legacy converter table.)

To accept None, add NoneType to this set.

**bitwise**

Only supported for unsigned integers. The native integer value of this Python argument will be written to the parameter without any range checking, even for negative values.

**converter**

Only supported by the object converter. Specifies the name of a C “converter function” to use to convert this object to a native type.

**encoding**

Only supported for strings. Specifies the encoding to use when converting this string from a Python str (Unicode) value into a C char * value.
subclass_of
Only supported for the object converter. Requires that the Python value be a subclass of a Python type, as expressed in C.

type
Only supported for the object and self converters. Specifies the C type that will be used to declare the variable. Default value is "PyObject *".

zeroes
Only supported for strings. If true, embedded NUL bytes ('\0') are permitted inside the value. The length of the string will be passed in to the impl function, just after the string parameter, as a parameter named <parameter_name>_length.

Please note, not every possible combination of arguments will work. Usually these arguments are implemented by specific PyArg_ParseTuple format units, with specific behavior. For example, currently you cannot call unsigned_short without also specifying bitwise=True. Although it's perfectly reasonable to think this would work, these semantics don't map to any existing format unit. So Argument Clinic doesn't support it. (Or, at least, not yet.)

Below is a table showing the mapping of legacy converters into real Argument Clinic converters. On the left is the legacy converter, on the right is the text you'd replace it with.

<table>
<thead>
<tr>
<th>Legacy Converter</th>
<th>Real Converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B'</td>
<td>unsigned_char(bitwise=True)</td>
</tr>
<tr>
<td>'b'</td>
<td>unsigned_char</td>
</tr>
<tr>
<td>'c'</td>
<td>char</td>
</tr>
<tr>
<td>'C'</td>
<td>int(accept={str})</td>
</tr>
<tr>
<td>Character</td>
<td>Python Type</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>'d'</td>
<td>double</td>
</tr>
<tr>
<td>'D'</td>
<td>Py_complex</td>
</tr>
<tr>
<td>'es'</td>
<td>str(encoding='name_of_encoding')</td>
</tr>
<tr>
<td>'es#'</td>
<td>str(encoding='name_of_encoding', zeroes=True)</td>
</tr>
<tr>
<td>'et'</td>
<td>str(encoding='name_of_encoding', accept={bytes, bytearray, str})</td>
</tr>
<tr>
<td>'et#'</td>
<td>str(encoding='name_of_encoding', accept={bytes, bytearray, str}, zeroes=True)</td>
</tr>
<tr>
<td>'f'</td>
<td>float</td>
</tr>
<tr>
<td>'h'</td>
<td>short</td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned_short(bitwise=True)</td>
</tr>
<tr>
<td>'i'</td>
<td>int</td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned_int(bitwise=True)</td>
</tr>
<tr>
<td>'k'</td>
<td>unsigned_long(bitwise=True)</td>
</tr>
<tr>
<td>'K'</td>
<td>unsigned_long_long(bitwise=True)</td>
</tr>
<tr>
<td>'l'</td>
<td>long</td>
</tr>
<tr>
<td>'L'</td>
<td>long long</td>
</tr>
<tr>
<td>'n'</td>
<td>Py_ssize_t</td>
</tr>
<tr>
<td>'O'</td>
<td>object</td>
</tr>
<tr>
<td>'O!'</td>
<td>object(subclass_of='&amp;PySomething_Type')</td>
</tr>
<tr>
<td>'O&amp;'</td>
<td>object(converter='name_of_c_function')</td>
</tr>
<tr>
<td>'p'</td>
<td>bool</td>
</tr>
<tr>
<td>'S'</td>
<td>PyBytesObject</td>
</tr>
<tr>
<td>'s'</td>
<td>str</td>
</tr>
<tr>
<td>'s#'</td>
<td>str(zeroes=True)</td>
</tr>
<tr>
<td>'s*'</td>
<td>Py_buffer(accept={buffer, str})</td>
</tr>
<tr>
<td>'U'</td>
<td>unicode</td>
</tr>
<tr>
<td>'u'</td>
<td>Py_UNICODE</td>
</tr>
<tr>
<td>'u#'</td>
<td>Py_UNICODE(zeroes=True)</td>
</tr>
<tr>
<td>'w*'</td>
<td>Py_buffer(accept={rwbuffer})</td>
</tr>
<tr>
<td>'Y'</td>
<td>PyByteArrayObject</td>
</tr>
<tr>
<td>'y'</td>
<td>str(accept={bytes})</td>
</tr>
<tr>
<td>'y#'</td>
<td>str(accept={robuffer}, zeroes=True)</td>
</tr>
<tr>
<td>'y*'</td>
<td>Py_buffer</td>
</tr>
<tr>
<td>'Z'</td>
<td>Py_UNICODE(accept={str, NoneType})</td>
</tr>
<tr>
<td>'Z#'</td>
<td>Py_UNICODE(accept={str, NoneType}, zeroes=True)</td>
</tr>
<tr>
<td>'z'</td>
<td>str(accept={str, NoneType})</td>
</tr>
<tr>
<td>'z#'</td>
<td>str(accept={str, NoneType}, zeroes=True)</td>
</tr>
<tr>
<td>'z*'</td>
<td>Py_buffer(accept={buffer, str, NoneType})</td>
</tr>
</tbody>
</table>

As an example, here's our sample pickle.Pickler.dump using the proper converter:

```c
/*[clinic input]
pickle.Pickler.dump
   obj: object
       The object to be pickled.
/
Write a pickled representation of obj to the open file
[clinic start generated code]*/
```
Argument Clinic will show you all the converters it has available. For each converter it'll show you all the parameters it accepts, along with the default value for each parameter. Just run `Tools/clinic/clinic.py --converters` to see the full list.

**Py_buffer**

When using the **Py_buffer** converter (or the `'s*', 'w*', 'y*', or 'z*' legacy converters), you **must** not call `PyBuffer_Release()` on the provided buffer. Argument Clinic generates code that does it for you (in the parsing function).

**Advanced converters**

Remember those format units you skipped for your first time because they were advanced? Here's how to handle those too.

The trick is, all those format units take arguments—either conversion functions, or types, or strings specifying an encoding. (But “legacy converters” don't support arguments. That's why we skipped them for your first function.) The argument you specified to the format unit is now an argument to the converter; this argument is either `converter` (for 0&), `subclass_of` (for 0!), or `encoding` (for all the format units that start with e).

When using `subclass_of`, you may also want to use the other custom argument for `object()`: `type`, which lets you set the type actually used for the parameter. For example, if you want to ensure that the object is a subclass of `PyUnicode_Type`, you probably want to use the converter `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`. 
One possible problem with using Argument Clinic: it takes away some possible flexibility for the format units starting with `e`. When writing a `PyArg_Parse` call by hand, you could theoretically decide at runtime what encoding string to pass in to `PyArg_ParseTuple()`. But now this string must be hard-coded at Argument-Clinic-preprocessing-time. This limitation is deliberate; it made supporting this format unit much easier, and may allow for future optimizations. This restriction doesn’t seem unreasonable; CPython itself always passes in static hard-coded encoding strings for parameters whose format units start with `e`.

**Parameter default values**

Default values for parameters can be any of a number of values. At their simplest, they can be string, int, or float literals:

```plaintext
foo: str = "abc"
bar: int = 123
bat: float = 45.6
```

They can also use any of Python’s built-in constants:

```plaintext
yep: bool = True
nope: bool = False
nada: object = None
```

There’s also special support for a default value of `NULL`, and for simple expressions, documented in the following sections.

**The NULL default value**

For string and object parameters, you can set them to `None` to indicate that there’s no default. However, that means the C variable
will be initialized to Py_None. For convenience's sakes, there's a special value called NULL for just this reason: from Python's perspective it behaves like a default value of None, but the C variable is initialized with NULL.

Expressions specified as default values

The default value for a parameter can be more than just a literal value. It can be an entire expression, using math operators and looking up attributes on objects. However, this support isn't exactly simple, because of some non-obvious semantics.

Consider the following example:

```python
go: Py_ssize_t = sys.maxsize - 1
```

`sys.maxsize` can have different values on different platforms. Therefore Argument Clinic can't simply evaluate that expression locally and hard-code it in C. So it stores the default in such a way that it will get evaluated at runtime, when the user asks for the function's signature.

What namespace is available when the expression is evaluated? It's evaluated in the context of the module the builtin came from. So, if your module has an attribute called “max_widgets”, you may simply use it:

```python
go: Py_ssize_t = max_widgets
```

If the symbol isn't found in the current module, it fails over to looking in `sys.modules`. That's how it can find `sys.maxsize` for example. (Since you don't know in advance what modules the user will load into
their interpreter, it's best to restrict yourself to modules that are preloaded by Python itself.)

Evaluating default values only at runtime means Argument Clinic can’t compute the correct equivalent C default value. So you need to tell it explicitly. When you use an expression, you must also specify the equivalent expression in C, using the `c_default` parameter to the converter:

```python
foo: Py_ssize_t(c_default="PY_SSIZE_T_MAX - 1") = sys.maxsize - 1
```

Another complication: Argument Clinic can't know in advance whether or not the expression you supply is valid. It parses it to make sure it looks legal, but it can't actually know. You must be very careful when using expressions to specify values that are guaranteed to be valid at runtime!

Finally, because expressions must be representable as static C values, there are many restrictions on legal expressions. Here’s a list of Python features you’re not permitted to use:

- Function calls.
- Inline if statements (3 if foo else 5).
- Automatic sequence unpacking (*[1, 2, 3]).
- List/set/dict comprehensions and generator expressions.
- Tuple/list/set/dict literals.

Using a return converter

By default the impl function Argument Clinic generates for you returns `PyObject *`. But your C function often computes some C type, then converts it into the `PyObject *` at the last moment. Argument Clinic
handles converting your inputs from Python types into native C types—why not have it convert your return value from a native C type into a Python type too?

That’s what a “return converter” does. It changes your impl function to return some C type, then adds code to the generated (non-impl) function to handle converting that value into the appropriate PyObject*.

The syntax for return converters is similar to that of parameter converters. You specify the return converter like it was a return annotation on the function itself. Return converters behave much the same as parameter converters; they take arguments, the arguments are all keyword-only, and if you’re not changing any of the default arguments you can omit the parentheses.

(If you use both "as" and a return converter for your function, the "as" should come before the return converter.)

There’s one additional complication when using return converters: how do you indicate an error has occurred? Normally, a function returns a valid (non-NULL) pointer for success, and NULL for failure. But if you use an integer return converter, all integers are valid. How can Argument Clinic detect an error? Its solution: each return converter implicitly looks for a special value that indicates an error. If you return that value, and an error has been set (PyErr_Occurred() returns a true value), then the generated code will propagate the error. Otherwise it will encode the value you return like normal.

Currently Argument Clinic supports only a few return converters:

\begin{verbatim}
bool
int
\end{verbatim}
None of these take parameters. For the first three, return -1 to indicate error. For DecodeFSDefault, the return type is `const char *`; return a NULL pointer to indicate an error.

(There's also an experimental NoneType converter, which lets you return `Py_None` on success or `NULL` on failure, without having to increment the reference count on `Py_None`. I'm not sure it adds enough clarity to be worth using.)

To see all the return converters Argument Clinic supports, along with their parameters (if any), just run `Tools/clinic/clinic.py --converters` for the full list.

**Cloning existing functions**

If you have a number of functions that look similar, you may be able to use Clinic's “clone” feature. When you clone an existing function, you reuse:

- its parameters, including
  - their names,
  - their converters, with all parameters,
  - their default values,
  - their per-parameter docstrings,
  - their *kind* (whether they’re positional only, positional or
keyword, or keyword only), and

- its return converter.

The only thing not copied from the original function is its docstring; the syntax allows you to specify a new docstring.

Here's the syntax for cloning a function:

```c
/*[clinic input]
module.class.new_function [as c_basename] = module.class.existing_function

Docstring for new_function goes here.
[clinic start generated code]*/
```

(The functions can be in different modules or classes. I wrote `module.class` in the sample just to illustrate that you must use the full path to both functions.)

Sorry, there's no syntax for partially-cloning a function, or cloning a function then modifying it. Cloning is an all-or nothing proposition.

Also, the function you are cloning from must have been previously defined in the current file.

**Calling Python code**

The rest of the advanced topics require you to write Python code which lives inside your C file and modifies Argument Clinic's runtime state. This is simple: you simply define a Python block.

A Python block uses different delimiter lines than an Argument Clinic function block. It looks like this:
All the code inside the Python block is executed at the time it’s parsed. All text written to stdout inside the block is redirected into the “output” after the block.

As an example, here’s a Python block that adds a static integer variable to the C code:

```python
/*[python input]
print('static int __ignored_unused_variable__ = 0;')
[python start generated code]*/
static int __ignored_unused_variable__ = 0;
/*[python checksum:...]*/
```

Using a “self converter”

Argument Clinic automatically adds a “self” parameter for you using a default converter. It automatically sets the type of this parameter to the “pointer to an instance” you specified when you declared the type. However, you can override Argument Clinic’s converter and specify one yourself. Just add your own self parameter as the first parameter in a block, and ensure that its converter is an instance of self_converter or a subclass thereof.

What’s the point? This lets you override the type of self, or give it a different default name.

How do you specify the custom type you want to cast self to? If you only have one or two functions with the same type for self, you can
directly use Argument Clinic's existing `self` converter, passing in the type you want to use as the `type` parameter:

```c
/*[clinic input]
.pickle.Pickler.dump

    self: self(type="PicklerObject *")
    obj: object
/

Write a pickled representation of the given object to
[clinic start generated code]*/
```

On the other hand, if you have a lot of functions that will use the same type for `self`, it’s best to create your own converter, subclassing `self_converter` but overwriting the `type` member:

```python
/*[python input]
class PicklerObject_converter(self_converter):
    type = "PicklerObject *
[python start generated code]*/

/*[clinic input]
.pickle.Pickler.dump

    self: PicklerObject
    obj: object
/

Write a pickled representation of the given object to
[clinic start generated code]*/
```
Writing a custom converter

As we hinted at in the previous section... you can write your own converters! A converter is simply a Python class that inherits from CConverter. The main purpose of a custom converter is if you have a parameter using the 0& format unit—parsing this parameter means calling a PyArg_ParseTuple() “converter function”.

Your converter class should be named *something*_converter. If the name follows this convention, then your converter class will be automatically registered with Argument Clinic; its name will be the name of your class with the _converter suffix stripped off. (This is accomplished with a metaclass.)

You shouldn't subclass CConverter.__init__. Instead, you should write a converter_init() function. converter_init() always accepts a self parameter; after that, all additional parameters must be keyword-only. Any arguments passed in to the converter in Argument Clinic will be passed along to your converter_init().

There are some additional members of CConverter you may wish to specify in your subclass. Here's the current list:

**type**

The C type to use for this variable. type should be a Python string specifying the type, e.g. int. If this is a pointer type, the type string should end with '*'.

**default**

The Python default value for this parameter, as a Python value. Or the magic value unspecified if there is no default.
default as it should appear in Python code, as a string. Or None if there is no default.

c_default
default as it should appear in C code, as a string. Or None if there is no default.

c_ignored_default

The default value used to initialize the C variable when there is no default, but not specifying a default may result in an “uninitialized variable” warning. This can easily happen when using option groups—although properly-written code will never actually use this value, the variable does get passed in to the impl, and the C compiler will complain about the “use” of the uninitialized value. This value should always be a non-empty string.

converter
The name of the C converter function, as a string.

impl_by_reference
A boolean value. If true, Argument Clinic will add a & in front of the name of the variable when passing it into the impl function.

parse_by_reference
A boolean value. If true, Argument Clinic will add a & in front of the name of the variable when passing it into PyArg_ParseTuple().

Here's the simplest example of a custom converter, from Modules/zlibmodule.c:

/*[python input]*/

class ssize_t_converter(CConverter):
    type = 'Py_ssize_t'
    converter = 'ssize_t_converter'

[python start generated code]*/
This block adds a converter to Argument Clinic named `ssize_t`. Parameters declared as `ssize_t` will be declared as type `Py_ssize_t`, and will be parsed by the 'O&' format unit, which will call the `ssize_t_converter` converter function. `ssize_t` variables automatically support default values.

More sophisticated custom converters can insert custom C code to handle initialization and cleanup. You can see more examples of custom converters in the CPython source tree; grep the C files for the string `CConverter`.

**Writing a custom return converter**

Writing a custom return converter is much like writing a custom converter. Except it’s somewhat simpler, because return converters are themselves much simpler.

Return converters must subclass `CReturnConverter`. There are no examples yet of custom return converters, because they are not widely used yet. If you wish to write your own return converter, please read `Tools/clinic/clinic.py`, specifically the implementation of `CReturnConverter` and all its subclasses.

**METH_O and METH_NOARGS**

To convert a function using `METH_O`, make sure the function's single argument is using the `object` converter, and mark the arguments as positional-only:
To convert a function using METH_NOARGS, just don’t specify any arguments.

You can still use a self converter, a return converter, and specify a type argument to the object converter for METH_O.

**tp_new and tp_init functions**

You can convert tp_new and tp_init functions. Just name them __new__ or __init__ as appropriate. Notes:

- The function name generated for __new__ doesn’t end in __new__ like it would by default. It’s just the name of the class, converted into a valid C identifier.
- No PyMethodDef #define is generated for these functions.
- __init__ functions return int, not PyObject *.
- Use the docstring as the class docstring.
- Although __new__ and __init__ functions must always accept both the args and kwargs objects, when converting you may specify any signature for these functions that you like. (If your function doesn’t support keywords, the parsing function generated will throw an exception if it receives any.)

**Changing and redirecting Clinic’s output**

It can be inconvenient to have Clinic’s output interspersed with your
conventional hand-edited C code. Luckily, Clinic is configurable: you can buffer up its output for printing later (or earlier!), or write its output to a separate file. You can also add a prefix or suffix to every line of Clinic's generated output.

While changing Clinic's output in this manner can be a boon to readability, it may result in Clinic code using types before they are defined, or your code attempting to use Clinic-generated code before it is defined. These problems can be easily solved by rearranging the declarations in your file, or moving where Clinic's generated code goes. (This is why the default behavior of Clinic is to output everything into the current block; while many people consider this hampers readability, it will never require rearranging your code to fix definition-before-use problems.)

Let's start with defining some terminology:

**field**

A field, in this context, is a subsection of Clinic's output. For example, the `#define` for the `PyMethodDef` structure is a field, called `methoddef_define`. Clinic has seven different fields it can output per function definition:

```
docstring_prototype
docstring_definition
methoddef_define
impl_prototype
parser_prototype
parser_definition
impl_definition
```

All the names are of the form "<a>_<b>"", where "<a>" is the semantic object represented (the parsing function, the impl
function, the docstring, or the methoddef structure) and "<b>" represents what kind of statement the field is. Field names that end in "_prototype" represent forward declarations of that thing, without the actual body/data of the thing; field names that end in "_definition" represent the actual definition of the thing, with the body/data of the thing. ("methoddef" is special, it's the only one that ends with "_define", representing that it's a preprocessor #define.)

destination

A destination is a place Clinic can write output to. There are five built-in destinations:

block

The default destination: printed in the output section of the current Clinic block.

buffer

A text buffer where you can save text for later. Text sent here is appended to the end of any existing text. It's an error to have any text left in the buffer when Clinic finishes processing a file.

file

A separate "clinic file" that will be created automatically by Clinic. The filename chosen for the file is {basename}.clinic{extension}, where basename and extension were assigned the output from os.path.splitext() run on the current file. (Example: the file destination for _pickle.c would be written to _pickle.clinic.c.)

Important: When using a file destination, you must check in the generated file!
two-pass
A buffer like buffer. However, a two-pass buffer can only be dumped once, and it prints out all text sent to it during all processing, even from Clinic blocks after the dumping point.

suppress
The text is suppressed—thrown away.

Clinic defines five new directives that let you reconfigure its output.

The first new directive is dump:

dump <destination>

This dumps the current contents of the named destination into the output of the current block, and empties it. This only works with buffer and two-pass destinations.

The second new directive is output. The most basic form of output is like this:

output <field> <destination>

This tells Clinic to output field to destination. output also supports a special meta-destination, called everything, which tells Clinic to output all fields to that destination.

output has a number of other functions:

output push
output pop
output preset <preset>

output push and output pop allow you to push and pop
configurations on an internal configuration stack, so that you can temporarily modify the output configuration, then easily restore the previous configuration. Simply push before your change to save the current configuration, then pop when you wish to restore the previous configuration.

**output preset** sets Clinic’s output to one of several built-in preset configurations, as follows:

```plaintext
block
Clinic’s original starting configuration. Writes everything immediately after the input block.

Supress the `parser_prototype` and `docstring_prototype`, write everything else to `block`.
```

```plaintext
file
Designed to write everything to the “clinic file” that it can. You then `#include` this file near the top of your file. You may need to rearrange your file to make this work, though usually this just means creating forward declarations for various `typedef` and `PyTypeObject` definitions.

Supress the `parser_prototype` and `docstring_prototype`, write the `impl_definition` to `block`, and write everything else to `file`.
```

The default filename is `${dirname}/clinic/{basename}.h`.

```plaintext
buffer
Save up most of the output from Clinic, to be written into your file near the end. For Python files implementing modules or
```
builtin types, it's recommended that you dump the buffer just above the static structures for your module or builtin type; these are normally very near the end. Using buffer may require even more editing than file, if your file has static PyMethodDef arrays defined in the middle of the file.

Suppress the parser_prototype, impl_prototype, and docstring_prototype, write the impl_definition to block, and write everything else to file.

two-pass
Similar to the buffer preset, but writes forward declarations to the two-pass buffer, and definitions to the buffer. This is similar to the buffer preset, but may require less editing than buffer. Dump the two-pass buffer near the top of your file, and dump the buffer near the end just like you would when using the buffer preset.

Suppresses the impl_prototype, write the impl_definition to block, write docstring_prototype, methoddef_define, and parser_prototype to two-pass, write everything else to buffer.

partial-buffer
Similar to the buffer preset, but writes more things to block, only writing the really big chunks of generated code to buffer. This avoids the definition-before-use problem of buffer completely, at the small cost of having slightly more stuff in the block's output. Dump the buffer near the end, just like you would when using the buffer preset.
Suppresses the `impl_prototype`, write the `docstring_definition` and `parser_definition` to `buffer`, write everything else to `block`.

The third new directive is `destination`:

```
destination <name> <command> [...]  
```

This performs an operation on the destination named `<name>`.

There are two defined subcommands: `new` and `clear`.

The `new` subcommand works like this:

```
destination <name> new <type> 
```

This creates a new destination with name `<name>` and type `<type>`.

There are five destination types:

- `suppress`
  - Throws the text away.
- `block`
  - Writes the text to the current block. This is what Clinic originally did.
- `buffer`
  - A simple text buffer, like the “buffer” builtin destination above.
- `file`
  - A text file. The file destination takes an extra argument, a template to use for building the filename, like so:

```
destination <name> new <type> <file_template>
```
The template can use three strings internally that will be replaced by bits of the filename:

{path}
   The full path to the file, including directory and full filename.

{dirname}
   The name of the directory the file is in.

{basename}
   Just the name of the file, not including the directory.

{basename_root}
   Basename with the extension clipped off (everything up to but not including the last ‘.’).

{basename_extension}
   The last ‘.’ and everything after it. If the basename does not contain a period, this will be the empty string.

If there are no periods in the filename, {basename} and {filename} are the same, and {extension} is empty. “{basename}{extension}” is always exactly the same as “{filename}”.

**two-pass**

A two-pass buffer, like the “two-pass” builtin destination above.

The **clear** subcommand works like this:

```
destination <name> clear
```

It removes all the accumulated text up to this point in the destination. (I don’t know what you’d need this for, but I thought maybe it’d be useful while someone’s experimenting.)
The fourth new directive is `set`:

```
set line_prefix "string"
set line_suffix "string"
```

`set` lets you set two internal variables in Clinic. `line_prefix` is a string that will be prepended to every line of Clinic’s output; `line_suffix` is a string that will be appended to every line of Clinic’s output.

Both of these support two format strings:

- `{block comment start}`
  - Turns into the string `/*`, the start-comment text sequence for C files.
- `{block comment end}`
  - Turns into the string `*/`, the end-comment text sequence for C files.

The final new directive is one you shouldn’t need to use directly, called `preserve`:

```
preserve
```

This tells Clinic that the current contents of the output should be kept, unmodified. This is used internally by Clinic when dumping output into file files; wrapping it in a Clinic block lets Clinic use its existing checksum functionality to ensure the file was not modified by hand before it gets overwritten.

**The #ifdef trick**
If you’re converting a function that isn’t available on all platforms, there’s a trick you can use to make life a little easier. The existing code probably looks like this:

```c
#ifdef HAVE_FUNCTIONNAME
static module_functionname(...) 
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

And then in the `PyMethodDef` structure at the bottom the existing code will have:

```c
#ifdef HAVE_FUNCTIONNAME
{'functionname', ...
},
#endif /* HAVE_FUNCTIONNAME */
```

In this scenario, you should enclose the body of your impl function inside the `#ifdef`, like so:

```c
#ifdef HAVE_FUNCTIONNAME
/*[clinic input]
module.functionname
...
[clinic start generated code]*/
static module_functionname(...) 
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

Then, remove those three lines from the `PyMethodDef` structure, replacing them with the macro Argument Clinic generated:
(You can find the real name for this macro inside the generated code. Or you can calculate it yourself: it's the name of your function as defined on the first line of your block, but with periods changed to underscores, uppercased, and "_METHODDEF" added to the end.)

Perhaps you’re wondering: what if HAVE_FUNCTIONNAME isn’t defined? The MODULE_FUNCTIONNAME_METHODDEF macro won’t be defined either!

Here’s where Argument Clinic gets very clever. It actually detects that the Argument Clinic block might be deactivated by the #ifdef. When that happens, it generates a little extra code that looks like this:

```
#ifdef MODULE_FUNCTIONNAME_METHODDEF
  #define MODULE_FUNCTIONNAME_METHODDEF
#endif /* !defined(MODULE_FUNCTIONNAME_METHODDEF) */
```

That means the macro always works. If the function is defined, this turns into the correct structure, including the trailing comma. If the function is undefined, this turns into nothing.

However, this causes one ticklish problem: where should Argument Clinic put this extra code when using the “block” output preset? It can’t go in the output block, because that could be deactivated by the #ifdef. (That’s the whole point!)

In this situation, Argument Clinic writes the extra code to the “buffer” destination. This may mean that you get a complaint from Argument Clinic:

Warning in file "Modules/posixmodule.c" on line 1235
When this happens, just open your file, find the `dump buffer` block that Argument Clinic added to your file (it'll be at the very bottom), then move it above the `PyMethodDef` structure where that macro is used.

### Using Argument Clinic in Python files

It’s actually possible to use Argument Clinic to preprocess Python files. There’s no point to using Argument Clinic blocks, of course, as the output wouldn’t make any sense to the Python interpreter. But using Argument Clinic to run Python blocks lets you use Python as a Python preprocessor!

Since Python comments are different from C comments, Argument Clinic blocks embedded in Python files look slightly different. They look like this:

```python
#/*[python input]
# print("def foo(): pass")
#[python start generated code]*/
def foo(): pass
#/*[python checksum:...]*/
```
Instrumenting CPython with DTrace and SystemTap

| author: | David Malcolm |
|author: | Łukasz Langa |

DTrace and SystemTap are monitoring tools, each providing a way to inspect what the processes on a computer system are doing. They both use domain-specific languages allowing a user to write scripts which:

- filter which processes are to be observed
- gather data from the processes of interest
- generate reports on the data

As of Python 3.6, CPython can be built with embedded “markers”, also known as “probes”, that can be observed by a DTrace or SystemTap script, making it easier to monitor what the CPython processes on a system are doing.

**CPython implementation detail:** DTrace markers are implementation details of the CPython interpreter. No guarantees are made about probe compatibility between versions of CPython. DTrace scripts can stop working or work incorrectly without warning when changing CPython versions.
Enabling the static markers

macOS comes with built-in support for DTrace. On Linux, in order to build CPython with the embedded markers for SystemTap, the SystemTap development tools must be installed.

On a Linux machine, this can be done via:

```sh
$ yum install systemtap-sdt-devel
```
or:

```sh
$ sudo apt-get install systemtap-sdt-dev
```

CPython must then be configured `--with-dtrace`:

```
checking for --with-dtrace... yes
```

On macOS, you can list available DTrace probes by running a Python process in the background and listing all probes made available by the Python provider:

```
$ python3.6 -q &
$ sudo dtrace -l -P python$!
```

```
ID	PROVIDER	MODULE		FUNCTION	NAME
29564 python18035 python3.6 _PyEval_EvalFrameDefault function-entry
29565 python18035 python3.6 dtrace_function_entry function-entry
29566 python18035 python3.6 _PyEval_EvalFrameDefault function-return
29567 python18035 python3.6 dtrace_function_return function-return
29568 python18035 python3.6 collect gc-done
29569 python18035 python3.6 collect gc-start
29570 python18035 python3.6 _PyEval_EvalFrameDefault line
29571 python18035 python3.6 maybe_dtrace_line line
```
On Linux, you can verify if the SystemTap static markers are present in the built binary by seeing if it contains a “.note.stapsdt” section.

```
$ readelf -S ./python | grep .note.stapsdt
[30] .note.stapsdt    NOTE  0000000000000000
```

If you've built Python as a shared library (with `--enable-shared`), you need to look instead within the shared library. For example:

```
$ readelf -S libpython3.3dm.so.1.0 | grep .note.stapsdt
[29] .note.stapsdt    NOTE  0000000000000000
```

Sufficiently modern readelf can print the metadata:

```
$ readelf -n ./python

Displaying notes found at file offset 0x00000254 with length 0x00000020:

<table>
<thead>
<tr>
<th>Owner</th>
<th>Data size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>0x00000010</td>
<td>NT_GNU_ABI_TAG (ABI version tag)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OS: Linux, ABI: 2.6.32</td>
</tr>
</tbody>
</table>

Displaying notes found at file offset 0x00000274 with length 0x00000024:

<table>
<thead>
<tr>
<th>Owner</th>
<th>Data size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>0x00000014</td>
<td>NT_GNU_BUILD_ID (unique build ID bitstring)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Build ID: df924a2b08a7e89f6e11251d4602022977</td>
</tr>
</tbody>
</table>

Displaying notes found at file offset 0x002d6c30 with length 0x00000144:

<table>
<thead>
<tr>
<th>Owner</th>
<th>Data size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stapsdt</td>
<td>0x00000031</td>
<td>NT_STAPSDT (SystemTap probe descriptors)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Provider: python</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name: gc__start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location: 0x00000000004371c3, Base: 0x00000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arguments: -4@%ebx</td>
</tr>
</tbody>
</table>
```
The above metadata contains information for SystemTap describing how it can patch strategically-placed machine code instructions to enable the tracing hooks used by a SystemTap script.
Static DTrace probes

The following example DTrace script can be used to show the call/return hierarchy of a Python script, only tracing within the invocation of a function called “start”. In other words, import-time function invocations are not going to be listed:

```c
self int indent;

python$target:::function-entry
/copyinstr(arg1) == "start"/
{
    self->trace = 1;
}

python$target:::function-entry
/self->trace/
{
    printf("%d\t%*s:", timestamp, 15, probename);
    printf("%*s", self->indent, "");
    printf("%s:%s:%d\n", basename(copyinstr(arg0)),
            self->indent++;
}

python$target:::function-return
/self->trace/
{
    self->indent--;
    printf("%d\t%*s:", timestamp, 15, probename);
    printf("%*s", self->indent, "");
    printf("%s:%s:%d\n", basename(copyinstr(arg0)),
            self->indent++;
}

python$target:::function-return
/copyinstr(arg1) == "start"/
```
It can be invoked like this:

```
$ sudo dtrace -q -s call_stack.d -c "python3.6 script.py"
```

The output looks like this:

```
156641360502280  function-entry:call_stack.py:start:23
156641360518804  function-entry: call_stack.py:function_1:1
156641360532797  function-entry: call_stack.py:function_3:9
156641360546807  function-return: call_stack.py:function_3:10
156641360563367  function-return: call_stack.py:function_1:2
156641360578365  function-entry: call_stack.py:function_2:5
156641360591757  function-entry: call_stack.py:function_1:1
156641360605556  function-entry: call_stack.py:function_3:9
156641360617482  function-return: call_stack.py:function_3:10
156641360629814  function-return: call_stack.py:function_1:2
156641360642285  function-return: call_stack.py:function_2:6
156641360656770  function-entry: call_stack.py:function_3:9
156641360669707  function-return: call_stack.py:function_3:10
156641360687853  function-entry: call_stack.py:function_4:13
156641360700719  function-return: call_stack.py:function_4:14
156641360719640  function-return: call_stack.py:function_5:18
156641360732567  function-return: call_stack.py:function_5:21
156641360747370  function-return:call_stack.py:start:28
```
Static SystemTap markers

The low-level way to use the SystemTap integration is to use the static markers directly. This requires you to explicitly state the binary file containing them.

For example, this SystemTap script can be used to show the call/return hierarchy of a Python script:

```c
probe process("python").mark("function__entry") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;

    printf("%s => %s in %s:%d\n",
           thread_indent(1), funcname, filename, lineno);
}

probe process("python").mark("function__return") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;

    printf("%s <= %s in %s:%d\n",
           thread_indent(-1), funcname, filename, lineno);
}
```

It can be invoked like this:

```bash
$ stap \\
    show-call-hierarchy.stp \\
    -c "./python test.py"
```

The output looks like this:
where the columns are:

- time in microseconds since start of script
- name of executable
- PID of process

and the remainder indicates the call/return hierarchy as the script executes.

For a \texttt{–enable-shared} build of CPython, the markers are contained within the libpython shared library, and the probe’s dotted path needs to reflect this. For example, this line from the above example:

\begin{verbatim}
probe process("python").mark("function__entry") {
\end{verbatim}

should instead read:

\begin{verbatim}
probe process("python").library("libpython3.6dm.so.1")
\end{verbatim}

(assuming a debug build of CPython 3.6)
Available static markers

**function__entry**(str *filename*, str *funcname*, int *lineno*)

This marker indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

The filename, function name, and line number are provided back to the tracing script as positional arguments, which must be accessed using `$arg1`, `$arg2`, `$arg3`:

- `$arg1`: (const char *) filename, accessible using `user_string($arg1)`
- `$arg2`: (const char *) function name, accessible using `user_string($arg2)`
- `$arg3`: int line number

**function__return**(str *filename*, str *funcname*, int *lineno*)

This marker is the converse of `function__entry()`, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

The arguments are the same as for `function__entry()`

**line**(str *filename*, str *funcname*, int *lineno*)

This marker indicates a Python line is about to be executed. It is the equivalent of line-by-line tracing with a Python profiler. It is not triggered within C functions.

The arguments are the same as for `function__entry()`

**gc__start**(int *generation*)
Fires when the Python interpreter starts a garbage collection cycle. 
*arg0* is the generation to scan, like `gc.collect()`.

**gc__done**(long *collected*)
Fires when the Python interpreter finishes a garbage collection cycle. *arg0* is the number of collected objects.

**import__find__load__start**(str *modulename*)
Fires before *importlib* attempts to find and load the module. *arg0* is the module name.

*New in version 3.7.*

**import__find__load__done**(str *modulename*, int *found*)
Fires after *importlib*’s *find_and_load* function is called. *arg0* is the module name, *arg1* indicates if module was successfully loaded.

*New in version 3.7.*
SystemTap Tapsets

The higher-level way to use the SystemTap integration is to use a “tapset”: SystemTap’s equivalent of a library, which hides some of the lower-level details of the static markers.

Here is a tapset file, based on a non-shared build of CPython:

```plaintext
/*
   Provide a higher-level wrapping around the function__entry
   and function__return markers:
\*/
probe python.function.entry = process("python").mark(
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
)

probe python.function.return = process("python").mark(
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
)
```

If this file is installed in SystemTap’s tapset directory (e.g. `/usr/share/systemtap/tapset`), then these additional probepoints become available:

```
python.function.entry(str filename, str funcname, int lineno, frameptr)
```
This probe point indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

```python
python.function.return(str filename, str funcname,
int lineno, frameptr)
```

This probe point is the converse of `python.function.return()`, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.
Examples

This SystemTap script uses the tapset above to more cleanly implement the example given above of tracing the Python function-call hierarchy, without needing to directly name the static markers:

```plaintext
probe python.function.entry
{    printf("%s => %s in %s:%d\n",
        thread_indent(1),
        funcname, filename, lineno);
}

probe python.function.return
{    printf("%s <= %s in %s:%d\n",
        thread_indent(-1),
        funcname, filename, lineno);
}
```

The following script uses the tapset above to provide a top-like view of all running CPython code, showing the top 20 most frequently-entered bytecode frames, each second, across the whole system:

```plaintext
global fn_calls;

probe python.function.entry
{    fn_calls[pid(), filename, funcname, lineno] += 1;
}

probe timer.ms(1000) {    printf("\033[2J\033[1;1H" /* clear screen */
                     printf("%6s %80s %6s %30s %6s\n",
                          "PID", "FILENAME", "LINE", "FUNCTION", "CALLs"
                     );
    foreach ([[pid, filename, funcname, lineno] in fn_calls]
```
printf("%6d %80s %6d %30s %6d\n",
    pid, filename, lineno, funcname,
    fn_calls[pid, filename, funcname, lineno],
}
delete fn_calls;
}
Contents

- General Python FAQ
  - General Information
    - What is Python?
    - What is the Python Software Foundation?
    - Are there copyright restrictions on the use of Python?
    - Why was Python created in the first place?
    - What is Python good for?
    - How does the Python version numbering scheme work?
    - How do I obtain a copy of the Python source?
    - How do I get documentation on Python?
    - I’ve never programmed before. Is there a Python tutorial?
    - Is there a newsgroup or mailing list devoted to Python?
    - How do I get a beta test version of Python?
    - How do I submit bug reports and patches for Python?
    - Are there any published articles about Python that I can reference?
    - Are there any books on Python?
    - Where in the world is www.python.org located?
    - Why is it called Python?
    - Do I have to like “Monty Python’s Flying Circus”?
  - Python in the real world
    - How stable is Python?
    - How many people are using Python?
    - Have any significant projects been done in Python?
    - What new developments are expected for Python in the future?
- Is it reasonable to propose incompatible changes to Python?
- Is Python a good language for beginning programmers?
General Information

What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later.

To find out more, start with The Python Tutorial. The Beginner’s Guide to Python links to other introductory tutorials and resources for learning Python.

What is the Python Software Foundation?

The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF’s mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF’s home page is at https://www.python.org/psf/.

Donations to the PSF are tax-exempt in the US. If you use Python and find it helpful, please contribute via the PSF donation page.

Are there copyright restrictions on the use of Python?
You can do anything you want with the source, as long as you leave
the copyrights in and display those copyrights in any documentation
about Python that you produce. If you honor the copyright rules, it's
OK to use Python for commercial use, to sell copies of Python in
source or binary form (modified or unmodified), or to sell products that
incorporate Python in some form. We would still like to know about all
commercial use of Python, of course.

See the PSF license page to find further explanations and a link to the
full text of the license.

The Python logo is trademarked, and in certain cases permission is
required to use it. Consult the Trademark Usage Policy for more
information.

Why was Python created in the first place?

Here's a very brief summary of what started it all, written by Guido van
Rossum:

I had extensive experience with implementing an interpreted
language in the ABC group at CWI, and from working with this
group I had learned a lot about language design. This is the origin
of many Python features, including the use of indentation for
statement grouping and the inclusion of very-high-level data types
(although the details are all different in Python).

I had a number of gripes about the ABC language, but also liked
many of its features. It was impossible to extend the ABC
language (or its implementation) to remedy my complaints – in
fact its lack of extensibility was one of its biggest problems. I had
some experience with using Modula-2+ and talked with the
designers of Modula-3 and read the Modula-3 report. Modula-3 is
the origin of the syntax and semantics used for exceptions, and some other Python features.

I was working in the Amoeba distributed operating system group at CWI. We needed a better way to do system administration than by writing either C programs or Bourne shell scripts, since Amoeba had its own system call interface which wasn't easily accessible from the Bourne shell. My experience with error handling in Amoeba made me acutely aware of the importance of exceptions as a programming language feature.

It occurred to me that a scripting language with a syntax like ABC but with access to the Amoeba system calls would fill the need. I realized that it would be foolish to write an Amoeba-specific language, so I decided that I needed a language that was generally extensible.

During the 1989 Christmas holidays, I had a lot of time on my hand, so I decided to give it a try. During the next year, while still mostly working on it in my own time, Python was used in the Amoeba project with increasing success, and the feedback from colleagues made me add many early improvements.

In February 1991, after just over a year of development, I decided to post to USENET. The rest is in the Misc/HISTORY file.

What is Python good for?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating
differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for The Python Standard Library to get an idea of what’s available. A wide variety of third-party extensions are also available. Consult the Python Package Index to find packages of interest to you.

How does the Python version numbering scheme work?

Python versions are numbered A.B.C or A.B. A is the major version number – it is only incremented for really major changes in the language. B is the minor version number, incremented for less earth-shattering changes. C is the micro-level – it is incremented for each bugfix release. See PEP 6 for more information about bugfix releases.

Not all releases are bugfix releases. In the run-up to a new major release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren’t yet finalized; it’s not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix. The suffix for an alpha version is “aN” for some small number N, the suffix for a beta version is “bN” for some small number N, and the suffix for a release candidate version is “cN” for some small number N. In other words, all versions labeled 2.0aN precede the versions
labeled 2.0bN, which precede versions labeled 2.0cN, and those precede 2.0.

You may also find version numbers with a “+” suffix, e.g. “2.2+”. These are unreleased versions, built directly from the CPython development repository. In practice, after a final minor release is made, the version is incremented to the next minor version, which becomes the “a0” version, e.g. “2.4a0”.

See also the documentation for sys.version, sys.hexversion, and sys.version_info.

How do I obtain a copy of the Python source?

The latest Python source distribution is always available from python.org, at https://www.python.org/downloads/. The latest development sources can be obtained at https://github.com/python/cpython/.

The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.

Consult the Getting Started section of the Python Developer’s Guide for more information on getting the source code and compiling it.

How do I get documentation on Python?

The standard documentation for the current stable version of Python is available at https://docs.python.org/3/. PDF, plain text, and
downloadable HTML versions are also available at https://docs.python.org/3/download.html.

The documentation is written in reStructuredText and processed by the Sphinx documentation tool. The reStructuredText source for the documentation is part of the Python source distribution.

I’ve never programmed before. Is there a Python tutorial?

There are numerous tutorials and books available. The standard documentation includes The Python Tutorial.

Consult the Beginner’s Guide to find information for beginning Python programmers, including lists of tutorials.

Is there a newsgroup or mailing list devoted to Python?

There is a newsgroup, comp.lang.python, and a mailing list, python-list. The newsgroup and mailing list are gatewayed into each other – if you can read news it’s unnecessary to subscribe to the mailing list. comp.lang.python is high-traffic, receiving hundreds of postings every day, and Usenet readers are often more able to cope with this volume.

Announcements of new software releases and events can be found in comp.lang.python.announce, a low-traffic moderated list that receives about five postings per day. It’s available as the python-announce mailing list.

More info about other mailing lists and newsgroups can be found at https://www.python.org/community/lists/.
How do I get a beta test version of Python?

Alpha and beta releases are available from https://www.python.org/downloads/. All releases are announced on the comp.lang.python and comp.lang.python.announce newsgroups and on the Python home page at https://www.python.org/; an RSS feed of news is available.

You can also access the development version of Python through Git. See The Python Developer’s Guide for details.

How do I submit bug reports and patches for Python?

To report a bug or submit a patch, please use the Roundup installation at https://bugs.python.org/.

You must have a Roundup account to report bugs; this makes it possible for us to contact you if we have follow-up questions. It will also enable Roundup to send you updates as we act on your bug. If you had previously used SourceForge to report bugs to Python, you can obtain your Roundup password through Roundup’s password reset procedure.

For more information on how Python is developed, consult the Python Developer’s Guide.

Are there any published articles about Python that I can reference?

It’s probably best to cite your favorite book about Python.
The very first article about Python was written in 1991 and is now quite outdated.


Are there any books on Python?

Yes, there are many, and more are being published. See the python.org wiki at https://wiki.python.org/moin/PythonBooks for a list.

You can also search online bookstores for “Python” and filter out the Monty Python references; or perhaps search for “Python” and “language”.

Where in the world is www.python.org located?

The Python project’s infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details here.

Why is it called Python?

When he began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

Do I have to like “Monty Python’s Flying Circus”? 

No, but it helps. :)

Python in the real world

How stable is Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. Currently there are usually around 18 months between major releases.

The developers issue “bugfix” releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it’s guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the Python download page. There are two production-ready version of Python: 2.x and 3.x, but the recommended one at this times is Python 3.x. Although Python 2.x is still widely used, it will not be maintained after January 1, 2020. Python 2.x was known for having more third-party libraries available, however, by the time of this writing, most of the widely used libraries support Python 3.x, and some are even dropping the Python 2.x support.

How many people are using Python?

There are probably tens of thousands of users, though it’s difficult to obtain an exact count.

Python is available for free download, so there are no sales figures, and it’s available from many different sites and packaged with many Linux distributions, so download statistics don’t tell the whole story.
The comp.lang.python newsgroup is very active, but not all Python users post to the group or even read it.

**Have any significant projects been done in Python?**

See [https://www.python.org/about/success](https://www.python.org/about/success) for a list of projects that use Python. Consulting the proceedings for past Python conferences will reveal contributions from many different companies and organizations.

High-profile Python projects include the Mailman mailing list manager and the Zope application server. Several Linux distributions, most notably Red Hat, have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

**What new developments are expected for Python in the future?**

See [https://www.python.org/dev/peps/](https://www.python.org/dev/peps/) for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled “Python X.Y Release Schedule”, where X.Y is a version that hasn’t been publicly released yet.

New development is discussed on the python-dev mailing list.

**Is it reasonable to propose incompatible changes**
to Python?

In general, no. There are already millions of lines of Python code around the world, so any change in the language that invalidates more than a very small fraction of existing programs has to be frowned upon. Even if you can provide a conversion program, there’s still the problem of updating all documentation; many books have been written about Python, and we don’t want to invalidate them all at a single stroke.

Providing a gradual upgrade path is necessary if a feature has to be changed. PEP 5 describes the procedure followed for introducing backward-incompatible changes while minimizing disruption for users.

Is Python a good language for beginning programmers?

Yes.

It is still common to start students with a procedural and statically typed language such as Pascal, C, or a subset of C++ or Java. Students may be better served by learning Python as their first language. Python has a very simple and consistent syntax and a large standard library and, most importantly, using Python in a beginning programming course lets students concentrate on important programming skills such as problem decomposition and data type design. With Python, students can be quickly introduced to basic concepts such as loops and procedures. They can probably even work with user-defined objects in their very first course.

For a student who has never programmed before, using a statically typed language seems unnatural. It presents additional complexity that
the student must master and slows the pace of the course. The students are trying to learn to think like a computer, decompose problems, design consistent interfaces, and encapsulate data. While learning to use a statically typed language is important in the long term, it is not necessarily the best topic to address in the students’ first programming course.

Many other aspects of Python make it a good first language. Like Java, Python has a large standard library so that students can be assigned programming projects very early in the course that do something. Assignments aren’t restricted to the standard four-function calculator and check balancing programs. By using the standard library, students can gain the satisfaction of working on realistic applications as they learn the fundamentals of programming. Using the standard library also teaches students about code reuse. Third-party modules such as PyGame are also helpful in extending the students’ reach.

Python’s interactive interpreter enables students to test language features while they’re programming. They can keep a window with the interpreter running while they enter their program’s source in another window. If they can’t remember the methods for a list, they can do something like this:

```python
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__del__
'dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__imul__', '__init__', '__iter__', '__le__', '__len__
'mul__', '__ne__', '__new__', '__reduce__', '__reduce_
'repr__', '__reversed__', '__rmul__', '__setattr__
'__sizeof__', '__str__', '__subclasshook__', 'append',
'copy', 'count', 'extend', 'index', 'insert', 'pop',
```
'reverse', 'sort']

```python
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

```python
>>> help(L.append)
Help on built-in function append:

append(...)  
    L.append(object) -> None -- append object to end
```

```python
>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as they are programming.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. PythonWin is a Windows-specific IDE. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult the Python wiki for a full list of Python editing environments.

If you want to discuss Python's use in education, you may be interested in joining the edu-sig mailing list.
Programming FAQ

Contents

- Programming FAQ
  - General Questions
    - Is there a source code level debugger with breakpoints, single-stepping, etc.?
    - Is there a tool to help find bugs or perform static analysis?
    - How can I create a stand-alone binary from a Python script?
    - Are there coding standards or a style guide for Python programs?
  - Core Language
    - Why am I getting an UnboundLocalError when the variable has a value?
    - What are the rules for local and global variables in Python?
    - Why do lambdas defined in a loop with different values all return the same result?
    - How do I share global variables across modules?
    - What are the “best practices” for using import in a module?
    - Why are default values shared between objects?
    - How can I pass optional or keyword parameters from one function to another?
    - What is the difference between arguments and parameters?
    - Why did changing list ‘y’ also change list ‘x’?
    - How do I write a function with output parameters (call by reference)?
- How do you make a higher order function in Python?
- How do I copy an object in Python?
- How can I find the methods or attributes of an object?
- How can my code discover the name of an object?
- What’s up with the comma operator’s precedence?
- Is there an equivalent of C’s “?:” ternary operator?
- Is it possible to write obfuscated one-liners in Python?
- What does the slash(/) in the parameter list of a function mean?
  - Numbers and strings
    - How do I specify hexadecimal and octal integers?
    - Why does -22 // 10 return -3?
    - How do I convert a string to a number?
    - How do I convert a number to a string?
    - How do I modify a string in place?
    - How do I use strings to call functions/methods?
    - Is there an equivalent to Perl’s chomp() for removing trailing newlines from strings?
    - Is there a scanf() or sscanf() equivalent?
    - What does ‘UnicodeDecodeError’ or ‘UnicodeEncodeError’ error mean?
  - Performance
    - My program is too slow. How do I speed it up?
    - What is the most efficient way to concatenate many strings together?
  - Sequences (Tuples/Lists)
    - How do I convert between tuples and lists?
    - What’s a negative index?
    - How do I iterate over a sequence in reverse order?
    - How do you remove duplicates from a list?
    - How do you make an array in Python?
    - How do I create a multidimensional list?
    - How do I apply a method to a sequence of objects?
    - Why does a_tuple[i] += ['item'] raise an exception when
the addition works?

- I want to do a complicated sort: can you do a Schwartzian Transform in Python?
- How can I sort one list by values from another list?

**Objects**

- What is a class?
- What is a method?
- What is self?
- How do I check if an object is an instance of a given class or of a subclass of it?
- What is delegation?
- How do I call a method defined in a base class from a derived class that overrides it?
- How can I organize my code to make it easier to change the base class?
- How do I create static class data and static class methods?
- How can I overload constructors (or methods) in Python?
- I try to use __spam and I get an error about _SomeClassName__spam.
- My class defines __del__ but it is not called when I delete the object.
- How do I get a list of all instances of a given class?
- Why does the result of id() appear to be not unique?

**Modules**

- How do I create a .pyc file?
- How do I find the current module name?
- How can I have modules that mutually import each other?
- __import__(‘x.y.z’) returns <module ‘x’>; how do I get z?
- When I edit an imported module and reimport it, the changes don’t show up. Why does this happen?
General Questions

Is there a source code level debugger with breakpoints, single-stepping, etc.?

Yes.

The pdb module is a simple but adequate console-mode debugger for Python. It is part of the standard Python library, and is documented in the Library Reference Manual. You can also write your own debugger by using the code for pdb as an example.

The IDLE interactive development environment, which is part of the standard Python distribution (normally available as Tools/scripts/idle), includes a graphical debugger.

PythonWin is a Python IDE that includes a GUI debugger based on pdb. The Pythonwin debugger colors breakpoints and has quite a few cool features such as debugging non-Pythonwin programs. Pythonwin is available as part of the Python for Windows Extensions project and as a part of the ActivePython distribution (see https://www.activestate.com/activepython).

Boa Constructor is an IDE and GUI builder that uses wxWidgets. It offers visual frame creation and manipulation, an object inspector, many views on the source like object browsers, inheritance hierarchies, doc string generated html documentation, an advanced debugger, integrated help, and Zope support.

Eric is an IDE built on PyQt and the Scintilla editing component.

Pydb is a version of the standard Python debugger pdb, modified for
use with DDD (Data Display Debugger), a popular graphical debugger front end. Pydb can be found at http://bashdb.sourceforge.net/pydb/ and DDD can be found at https://www.gnu.org/software/ddd.

There are a number of commercial Python IDEs that include graphical debuggers. They include:

- Wing IDE (https://wingware.com/)
- Komodo IDE (https://komodoide.com/)
- PyCharm (https://www.jetbrains.com/pycharm/)

Is there a tool to help find bugs or perform static analysis?

Yes.

PyChecker is a static analysis tool that finds bugs in Python source code and warns about code complexity and style. You can get PyChecker from http://pychecker.sourceforge.net/.

Pylint is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature. In addition to the bug checking that PyChecker performs, Pylint offers some additional features such as checking line length, whether variable names are well-formed according to your coding standard, whether declared interfaces are fully implemented, and more. https://docs.pylint.org/ provides a full list of Pylint’s features.

Static type checkers such as Mypy, Pyre, and Pytype can check type hints in Python source code.

How can I create a stand-alone binary from a Python script?
You don’t need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as `Tools/freeze`. It converts Python byte code to C arrays; a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

Obviously, freeze requires a C compiler. There are several other utilities which don't. One is Thomas Heller’s py2exe (Windows only) at http://www.py2exe.org/

Another tool is Anthony Tuininga’s cx_Freeze.

**Are there coding standards or a style guide for Python programs?**
Yes. The coding style required for standard library modules is documented as **PEP 8**.
Core Language

Why am I getting an UnboundLocalError when the variable has a value?

It can be a surprise to get the UnboundLocalError in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

This code:

```python
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

works, but this code:

```python
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

results in an UnboundLocalError:

```python
>>> foo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any
similarly named variable in the outer scope. Since the last statement in foo assigns a new value to x, the compiler recognizes it as a local variable. Consequently when the earlier print(x) attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it global:

```python
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```python
>>> print(x)
11
```

You can do a similar thing in a nested scope using the nonlocal keyword:

```python
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
```
What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

Why do lambdas defined in a loop with different values all return the same result?

Assume you use a for loop to define a few different lambdas (or even plain functions), e.g.:

```python
>>> squares = []
>>> for x in range(5):
...    squares.append(lambda: x**2)
```

This gives you a list that contains 5 lambdas that calculate $x^2$. You might expect that, when called, they would return, respectively, 0, 1,
4, 9, and 16. However, when you actually try you will see that they all return 16:

```python
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is 4, so all the functions now return `4**2`, i.e. 16. You can also verify this by changing the value of `x` and see how the results of the lambdas change:

```python
>>> x = 8
>>> squares[2]()
64
```

In order to avoid this, you need to save the values in variables local to the lambdas, so that they don’t rely on the value of the global `x`:

```python
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Here, `n=x` creates a new variable `n` local to the lambda and computed when the lambda is defined so that it has the same value that `x` had at that point in the loop. This means that the value of `n` will be 0 in the first lambda, 1 in the second, 2 in the third, and so on. Therefore each lambda will now return the correct result:

```python
>>> squares[2]()
```
Note that this behaviour is not peculiar to lambdas, but applies to regular functions too.

**How do I share global variables across modules?**

The canonical way to share information across modules within a single program is to create a special module (often called config or cfg). Just import the config module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

config.py:

```python
x = 0  # Default value of the 'x' configuration setting
```

mod.py:

```python
import config
config.x = 1
```

main.py:

```python
import config
import mod
print(config.x)
```

Note that using a module is also the basis for implementing the Singleton design pattern, for the same reason.
What are the “best practices” for using import in a module?

In general, don’t use `from modulename import *`. Doing so clutters the importer’s namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It’s good practice if you import modules in the following order:

1. standard library modules – e.g. `sys`, `os`, `getopt`, `re`
2. third-party library modules (anything installed in Python’s site-packages directory) – e.g. `mx.DateTime`, `ZODB`, `PIL.Image`, etc.
3. locally-developed modules

It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

Circular imports are fine where both modules use the “import <module>” form of import. They fail when the 2nd module wants to grab a name out of the first (“from module import name”) and the import is at the top level. That’s because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the
second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are platform-specific. In that case, it may not even be possible to import all of the modules at the top of the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it's necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

**Why are default values shared between objects?**

This type of bug commonly bites neophyte programmers. Consider this function:

```python
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, `mydict` contains a single item. The second time, `mydict` contains two items because when `foo()`
began executing, mydict starts out with an item already in it.

It is often expected that a function call creates new objects for default values. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

By definition, immutable objects such as numbers, strings, tuples, and None, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, it is good programming practice to not use mutable objects as default values. Instead, use None as the default value and inside the function, check if the parameter is None and create a new list/dictionary/whatever if it is. For example, don’t write:

```python
def foo(mydict={}):
    ...
```

but:

```python
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

This feature can be useful. When you have a function that’s time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called “memoizing”, and can be implemented like this:

```python
# Callers can only provide two parameters and option
```
```python
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result  # Store
defensive result
```

You could use a global variable containing a dictionary instead of the default value; it’s a matter of taste.

**How can I pass optional or keyword parameters from one function to another?**

Collect the arguments using the `*` and `**` specifiers in the function's parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using `*` and `**`:

```python
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

**What is the difference between arguments and parameters?**

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what types of arguments a
function can accept. For example, given the function definition:

```python
def func(foo, bar=None, **kwargs):
    pass
```

`foo`, `bar` and `kwargs` are parameters of `func`. However, when calling `func`, for example:

```python
func(42, bar=314, extra=somevar)
```

the values 42, 314, and `somevar` are arguments.

**Why did changing list ‘y’ also change list ‘x’?**

If you wrote code like:

```python
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

you might be wondering why appending an element to `y` changed `x` too.

There are two factors that produce this result:

1. Variables are simply names that refer to objects. Doing `y = x` doesn’t create a copy of the list – it creates a new variable `y` that refers to the same object `x` refers to. This means that there is only one object (the list), and both `x` and `y` refer to it.
2. Lists are **mutable**, which means that you can change their
After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

If we instead assign an immutable object to `x`:

```python
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating
6
>>> y
5
```

we can see that in this case `x` and `y` are not equal anymore. This is because integers are immutable, and when we do `x = x + 1` we are not mutating the int 5 by incrementing its value; instead, we are creating a new object (the int 6) and assigning it to `x` (that is, changing which object `x` refers to). After this assignment we have two objects (the ints 6 and 5) and two variables that refer to them (now refers to 6 but `y` still refers to 5).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily
diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (\texttt{a\_list += [1, 2, 3]} is equivalent to \texttt{a\_list.extend([1, 2, 3])} and mutates \texttt{a\_list}, whereas \texttt{some\_tuple += (1, 2, 3)} and \texttt{some\_int += 1} create new objects).

In other words:

- If we have a mutable object (\texttt{list}, \texttt{dict}, \texttt{set}, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (\texttt{str}, \texttt{int}, \texttt{tuple}, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the \texttt{is} operator, or the built-in function \texttt{id()}.

**How do I write a function with output parameters (call by reference)?**

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there’s no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number of ways.

1. By returning a tuple of the results:
```python
def func2(a, b):
a = 'new-value'  # a and b are local names
b = b + 1  # assigned to new objects
return a, b  # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print(x, y)  # output: new-value 100
```

This is almost always the clearest solution.

2. By using global variables. This isn’t thread-safe, and is not recommended.

3. By passing a mutable (changeable in-place) object:
```python
def func1(a):
a[0] = 'new-value'  # 'a' references a mutable list
a[1] = a[1] + 1  # changes a shared object

args = ['old-value', 99]
func1(args)
print(args[0], args[1])  # output: new-value 100
```

4. By passing in a dictionary that gets mutated:
```python
def func3(args):
    args['a'] = 'new-value'  # args is a mutable dictionary
    args['b'] = args['b'] + 1  # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print(args['a'], args['b'])
```
5. Or bundle up values in a class instance:

```python
class callByRef:
    def __init__(self, **args):
        for (key, value) in args.items():
            setattr(self, key, value)

def func4(args):
    args.a = 'new-value'  # args is a mutable
    args.b = args.b + 1   # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print(args.a, args.b)
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

**How do you make a higher order function in Python?**

You have two choices: you can use nested scopes or you can use callable objects. For example, suppose you wanted to define `linear(a, b)` which returns a function `f(x)` that computes the value `a*x+b`. Using nested scopes:

```python
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Or using a callable object:
```python
class linear:
    def __init__(self, a, b):
        self.a, self.b = a, b
    def __call__(self, x):
        return self.a * x + self.b
```

In both cases,

```python
taxes = linear(0.3, 2)
```
gives a callable object where `taxes(10e6) == 0.3 * 10e6 + 2`.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

```python
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Object can encapsulate state for several methods:

```python
class counter:
    value = 0
    def set(self, x):
        self.value = x
    def up(self):
        self.value = self.value + 1
    def down(self):
```

self.value = self.value - 1

```
count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Here `inc()`, `dec()` and `reset()` act like functions which share the same counting variable.

**How do I copy an object in Python?**

In general, try `copy.copy()` or `copy.deepcopy()` for the general case. Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a `copy()` method:

```
newdict = olddict.copy()
```

Sequences can be copied by slicing:

```
new_l = l[:]
```

**How can I find the methods or attributes of an object?**

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

**How can my code discover the name of an object?**
Generally speaking, it can’t, because objects don’t really have names. Essentially, assignment always binds a name to a value; The same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```python
>>> class A:
...     pass
... >>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<_main__.A object at 0x16D07CC>
>>> print(a)
<_main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name B the created instance is still reported as an instance of class A. However, it is impossible to say whether the instance’s name is a or b, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to “know the names” of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

In comp.lang.python, Fredrik Lundh once gave an excellent analogy in answer to this question:

The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn’t really care – so the only way to find out what it’s called is to ask all your neighbours (namespaces) if it’s their cat (object)…
....and don't be surprised if you'll find that it's known by many names, or no name at all!

What's up with the comma operator's precedence?

Comma is not an operator in Python. Consider this session:

```python
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```python
("a" in "b"), "a"
```

not:

```python
"a" in ("b", "a")
```

The same is true of the various assignment operators (=, += etc). They are not truly operators but syntactic delimiters in assignment statements.

Is there an equivalent of C’s “?:” ternary operator?

Yes, there is. The syntax is as follows:

```python
[on_true] if [expression] else [on_false]
```

```python
x, y = 50, 25
small = x if x < y else y
```
Before this syntax was introduced in Python 2.5, a common idiom was to use logical operators:

```
[expression] and [on_true] or [on_false]
```

However, this idiom is unsafe, as it can give wrong results when `on_true` has a false boolean value. Therefore, it is always better to use the `... if ... else ...` form.

**Is it possible to write obfuscated one-liners in Python?**

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, due to Ulf Bartelt:

```python
from functools import reduce

# Primes < 1000
print(list(filter(None, map(lambda y: y * reduce(lambda x, y=y:y%x, range(2, int(pow(y, 0.5) + 1))), 1),
# First 10 Fibonacci numbers
print(list(map(lambda x, f=lambda x, y: x, f: (f(x-1, f)+f(x-2, f(x, f)), range(10)))))

# Mandelbrot set
print((lambda Ru, Ro, Iu, Io, IM, Sx, Sy: reduce(lambda x, y: x, y + y, map(lambda x, xc=Ru, i=i, Sx=Sx, F=lambda xc, yc, x, y, k, f=lambda xc, yc, x, y, k, >=4.0) or 1+f(xc, yc, x*xc-y*yc, 2.0*x*yc, k-1, f):f(x, 64+F(Ru+x*(Ro-Ru)/Sx, yc, 0, 0, i)), range(Sx))):L(Iu+y*(
# __ __ ___/ __ __ ___/ | | | __ lines on screen
```
Don’t try this at home, kids!

What does the slash(/) in the parameter list of a function mean?

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally-usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `pow()` is a function that accepts positional-only parameters. Its documentation looks like this:

```python
>>> help(pow)
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
   Equivalent to x**y (with two arguments) or x**y % z
   Some types, such as ints, are able to use a more
   invoked using the three argument form.
```

The slash at the end of the parameter list means that all three parameters are positional-only. Thus, calling `pow()` with keyword arguments would lead to an error:

```python
>>> pow(x=3, y=4)
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pow() takes no keyword arguments

Note that as of this writing this is only documentational and no valid syntax in Python, although there is PEP 570, which proposes a syntax for position-only parameters in Python.
Numbers and strings

How do I specify hexadecimal and octal integers?

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase “o”. For example, to set the variable “a” to the octal value “10” (8 in decimal), type:

```python
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase “x”. Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```python
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

Why does -22 // 10 return -3?

It’s primarily driven by the desire that \( i \% j \) have the same sign as \( j \). If you want that, and also want:

\[
i = (i \div j) \times j + (i \% j)\]

then integer division has to return the floor. C also requires that identity
to hold, and then compilers that truncate \( i \div j \) need to make \( i \mod j \) have the same sign as \( i \).

There are few real use cases for \( i \mod j \) when \( j \) is negative. When \( j \) is positive, there are many, and in virtually all of them it’s more useful for \( i \mod j \) to be \( \geq 0 \). If the clock says 10 now, what did it say 200 hours ago? \(-190 \mod 12 == 2\) is useful; \(-190 \mod 12 == -10\) is a bug waiting to bite.

**How do I convert a string to a number?**

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to floating-point, e.g. `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int('0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python’s rules: a leading ‘0o’ indicates octal, and ‘0x’ indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because
Python does not allow leading ‘0’ in a decimal number (except ‘0’).

**How do I convert a number to a string?**

To convert, e.g., the number 144 to the string ‘144’, use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the Formatted string literals and Format String Syntax sections, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:3f}".format(1.0/3.0)` yields `'0.333'`.

**How do I modify a string in place?**

You can’t, because strings are immutable. In most situations, you should simply construct a new string from the various parts you want to assemble it from. However, if you need an object with the ability to modify in-place unicode data, try using an `io.StringIO` object or the `array` module:

```python
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
```
How do I use strings to call functions/methods?

There are various techniques.

- The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

```python
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- Use the built-in function `getattr()`:

```python
import foo
getattr(foo, 'bar')()
```

Note that `getattr()` works on any object, including classes, class instances, modules, and so on.
This is used in several places in the standard library, like this:

```python
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Use `locals()` or `eval()` to resolve the function name:

```python
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()
```

Note: Using `eval()` is slow and dangerous. If you don’t have absolute control over the contents of the string, someone could pass a string that resulted in an arbitrary function being executed.

**Is there an equivalent to Perl’s chomp() for removing trailing newlines from strings?**

You can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other
trailing whitespace. If the string S represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```python
>>> lines = ("line 1 \r\n"...
           "\r\n"
           "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Since this is typically only desired when reading text one line at a time, using S.rstrip() this way works well.

**Is there a scanf() or sscanf() equivalent?**

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional “sep” parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C’s `sscanf()` and better suited for the task.

**What does ‘UnicodeDecodeError’ or ‘UnicodeEncodeError’ error mean?**

See the Unicode HOWTO.
Performance

My program is too slow. How do I speed it up?

That’s a tough one, in general. First, here are a list of things to remember before diving further:

- Performance characteristics vary across Python implementations. This FAQ focuses on CPython.
- Behaviour can vary across operating systems, especially when talking about I/O or multi-threading.
- You should always find the hot spots in your program before attempting to optimize any code (see the profile module).
- Writing benchmark scripts will allow you to iterate quickly when searching for improvements (see the timeit module).
- It is highly recommended to have good code coverage (through unit testing or any other technique) before potentially introducing regressions hidden in sophisticated optimizations.

That being said, there are many tricks to speed up Python code. Here are some general principles which go a long way towards reaching acceptable performance levels:

- Making your algorithms faster (or changing to faster ones) can yield much larger benefits than trying to sprinkle micro-optimization tricks all over your code.
- Use the right data structures. Study documentation for the Built-in Types and the collections module.
- When the standard library provides a primitive for doing something, it is likely (although not guaranteed) to be faster than any alternative you may come up with. This is doubly true for primitives written in C, such as builtins and some extension types. For example, be sure to use either the list.sort() built-in
method or the related `sorted()` function to do sorting (and see the Sorting HOW TO for examples of moderately advanced usage).

- Abstractions tend to create indirections and force the interpreter to work more. If the levels of indirection outweigh the amount of useful work done, your program will be slower. You should avoid excessive abstraction, especially under the form of tiny functions or methods (which are also often detrimental to readability).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, Cython can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

See also: The wiki page devoted to performance tips.

What is the most efficient way to concatenate many strings together?

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, the recommended idiom is to place them into a list and call `str.join()` at the end:

```python
chunks = []
for s in my_strings:
    chunks.append(s)
```
result = ''.join(chunks)

(another reasonably efficient idiom is to use `io.StringIO`)

To accumulate many `bytes` objects, the recommended idiom is to extend a `bytearray` object using in-place concatenation (the `+=` operator):

```python
result = bytearray()
for b in my_bytes_objects:
    result += b
```
Sequences (Tuples/Lists)

How do I convert between tuples and lists?

The type constructor `tuple(seq)` converts any sequence (actually, any iterable) into a tuple with the same items in the same order.

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `('a', 'b', 'c')`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields `['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

What’s a negative index?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[::-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.
How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function, which is new in Python 2.4:

```python
for x in reversed(sequence):
    ...  # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

With Python 2.3, you can use an extended slice syntax:

```python
for x in sequence[::-1]:
    ...  # do something with x ...
```

How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

https://code.activestate.com/recipes/52560/

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```python
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
```
last = mylist[i]

If all elements of the list may be used as set keys (i.e. they are all hashable) this is often faster

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

**How do you make an array in Python?**

Use a list:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that the Numeric extensions and others define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate cons cells using tuples:

```
lisp_list = ("like", ("this", ("example", None))
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of lisp car is `lisp_list[0]` and the analogue of cdr is `lisp_list[1]`. Only do this if you're sure you really need to,
because it's usually a lot slower than using Python lists.

**How do I create a multidimensional list?**

You probably tried to make a multidimensional array like this:

```python
>>> A = [[None] * 2] * 3
```

This looks correct if you print it:

```python
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```python
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn’t create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```python
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:
\[ w, h = 2, 3 \]
\[ A = [[\text{None}] * w \text{ for } i \text{ in range}(h)] \]

Or, you can use an extension that provides a matrix datatype; NumPy is the best known.

**How do I apply a method to a sequence of objects?**

Use a list comprehension:

\[
\text{result} = [\text{obj.method()} \text{ for obj in mylist}]\]

**Why does a_tuple[i] += ['item'] raise an exception when the addition works?**

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a `list` and `+=` as our exemplar.

If you wrote:

```python
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```
The reason for the exception should be immediately clear: 1 is added to the object \texttt{a\_tuple[0]} points to (1), producing the result object, 2, but when we attempt to assign the result of the computation, 2, to element 0 of the tuple, we get an error because we can’t change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a\_tuple[0] + 1
>>> a\_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a\_tuple = (['foo'], 'bar')
>>> a\_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a\_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object
implements an \_\_iadd\_\_ magic method, it gets called when the +\= augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, \_\_iadd\_\_ is equivalent to calling extend on the list and returning the list. That’s why we say that for lists, +\= is a “shorthand” for list.extend:

```python
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

This is equivalent to:

```python
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to a_list. The end result of the assignment is a no-op, since it is a pointer to the same object that a_list was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```python
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The \_\_iadd\_\_ succeeds, and thus the list is extended, but even though result points to the same object that a_tuple[0] already points to, that final assignment still results in an error, because tuples are immutable.
I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its “sort value”. In Python, use the `key` argument for the `list.sort()` method:

```python
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

How can I sort one list by values from another list?

Merge them into an iterator of tuples, sort the resulting list, and then pick out the element you want.

```python
>>> list1 = ['what', 'I'm', 'sorting', 'by']
>>> list2 = ['something', 'else', 'to', 'sort']
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)

[('I'm', 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
```

An alternative for the last step is:

```python
>>> result = []
>>> for p in pairs: result.append(p[1])

['else', 'sort', 'to', 'something']
```

If you find this more legible, you might prefer to use this instead of the
final list comprehension. However, it is almost twice as slow for long lists. Why? First, the `append()` operation has to reallocate memory, and while it uses some tricks to avoid doing that each time, it still has to do it occasionally, and that costs quite a bit. Second, the expression “result.append” requires an extra attribute lookup, and third, there’s a speed reduction from having to make all those function calls.
Objects

What is a class?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic Mailbox class that provides basic accessor methods for a mailbox, and subclasses such as MboxMailbox, MaildirMailbox, OutlookMailbox that handle various specific mailbox formats.

What is a method?

A method is a function on some object x that you normally call as x.name(arguments...). Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

What is self?

Self is merely a conventional name for the first argument of a method. A method defined as meth(self, a, b, c) should be called as
x.meth(a, b, c) for some instance x of the class in which the definition occurs; the called method will think it is called as meth(x, a, b, c).

See also Why must ‘self’ be used explicitly in method definitions and calls?

How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python’s built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object’s class and doing a different thing based on what class it is. For example, if you have a function that does something:

```python
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes
and just call it:

```python
class Mailbox:
    def search(self):
        ...
        # code to search a mailbox

class Document:
    def search(self):
        ...
        # code to search a document

obj.search()
```

What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you’re interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```python
class UpperOut:
    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
```
return getattr(self._outfile, name)

Here the UpperOut class redefines the write() method to convert the argument string to uppercase before calling the underlying self.__outfile.write() method. All other methods are delegated to the underlying self.__outfile object. The delegation is accomplished via the __getattr__ method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a __setattr__() method too, and it must do so carefully. The basic implementation of __setattr__() is roughly equivalent to the following:

class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...

Most __setattr__() implementations must modify self.__dict__ to store local state for self without causing an infinite recursion.

How do I call a method defined in a base class from a derived class that overrides it?

Use the built-in super() function:

class Derived(Base):
    def meth(self):
For version prior to 3.0, you may be using classic classes: For a class definition such as
\begin{verbatim}
class Derived(Base):
    ...
\end{verbatim}
you can call method \texttt{meth()} defined in \texttt{Base} (or one of \texttt{Base}'s base classes) as
\texttt{Base.meth(self, arguments...}). Here, \texttt{Base.meth} is an unbound method, so you need to provide the \texttt{self} argument.

\textbf{How can I organize my code to make it easier to change the base class?}

You could define an alias for the base class, assign the real base class to it before your class definition, and use the alias throughout your class. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

\begin{verbatim}
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
    ...
\end{verbatim}

\textbf{How do I create static class data and static class methods?}

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value
to the attribute, you have to explicitly use the class name in the assignment:

class C:
    count = 0  # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count

c.count also refers to C.count for any c such that isinstance(c, C) holds, unless overridden by c itself or by some class on the base-class search path from c.__class__ back to C.

Caution: within a method of C, an assignment like self.count = 42 creates a new and unrelated instance named “count” in self’s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

C.count = 314

Static methods are possible:

class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:
def getcount():
    return C.count

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

How can I overload constructors (or methods) in Python?

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

In C++ you’d write

class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n";
}

In Python you have to write a single constructor that catches all cases using default arguments. For example:

class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)

This is not entirely equivalent, but close enough in practice.

You could also try a variable-length argument list, e.g.
def __init__(self, *args):
...

The same approach works for all method definitions.

I try to use __spam and I get an error about _SomeClassName__spam.

Variable names with double leading underscores are “mangled” to provide a simple but effective way to define class private variables. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with any leading underscores stripped.

This doesn’t guarantee privacy: an outside user can still deliberately access the “_classname__spam” attribute, and private values are visible in the object's __dict__. Many Python programmers never bother to use private variable names at all.

My class defines __del__ but it is not called when I delete the object.

There are several possible reasons for this.

The del statement does not necessarily call __del__() – it simply decrements the object’s reference count, and if this reaches zero __del__() is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python
runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your __del__() method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's __del__() methods are executed is arbitrary. You can run gc.collect() to force a collection, but there are pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit close() method on objects to be called whenever you're done with them. The close() method can then remove attributes that refer to subobjects. Don't call __del__() directly – __del__() should call close() and close() should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the weakref module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

Finally, if your __del__() method raises an exception, a warning message is printed to sys.stderr.

How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.
Why does the result of `id()` appear to be not unique?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000) # doctest: +SKIP
13901272
>>> id(2000) # doctest: +SKIP
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a) # doctest: +SKIP
13901272
>>> id(b) # doctest: +SKIP
13891296
```
Modules

How do I create a .pyc file?

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a .pyc file containing the compiled code should be created in a __pycache__ subdirectory of the directory containing the .py file. The .pyc file will have a filename that starts with the same name as the .py file, and ends with .pyc, with a middle component that depends on the particular python binary that created it. (See PEP 3147 for details.)

One reason that a .pyc file may not be created is a permissions problem with the directory containing the source file, meaning that the __pycache__ subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the PYTHONDONOTWRITEBYTECODE environment variable is set, creation of a .pyc file is automatic if you’re importing a module and Python has the ability (permissions, free space, etc…) to create a __pycache__ subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no .pyc will be created. For example, if you have a top-level module foo.py that imports another module xyz.py, when you run foo (by typing python foo.py as a shell command), a .pyc will be created for xyz because xyz is imported, but no .pyc file will be created for foo since foo.py isn't being imported.
If you need to create a `.pyc` file for `foo` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

The `py_compile` module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```python
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to a `__pycache__` subdirectory in the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

How do I find the current module name?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `['__main__']`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```python
def main():
    print('Running test...')
```
How can I have modules that mutually import each other?

Suppose you have the following modules:

foo.py:

```python
from bar import bar_var
foo_var = 1
```

bar.py:

```python
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports foo
- Empty globals for foo are created
- foo is compiled and starts executing
- foo imports bar
- Empty globals for bar are created
- bar is compiled and starts executing
- bar imports foo (which is a no-op since there already is a module named foo)
- bar.foo_var = foo.foo_var

The last step fails, because Python isn’t done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.
The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>..<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- `import` statements
- active code (including globals that are initialized from imported values).

van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

```
__import__('x.y.z') returns <module 'x'>; how do I get z?
```

Consider using the convenience function `import_module()` from `importlib` instead:
When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```python
import importlib
import modname
importlib.reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules containing statements like

```python
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will not be updated to use the new class definition. This can result in the following paradoxical behaviour:

```bash
>>> import importlib
>>> import cls
>>> c = cls.C()  # Create an instance
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)  # isinstance is false
False
```
The nature of the problem is made clear if you print out the “identity” of the class objects:

```python
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```
Design and History FAQ

Contents

- Design and History FAQ
  - Why does Python use indentation for grouping of statements?
  - Why am I getting strange results with simple arithmetic operations?
  - Why are floating-point calculations so inaccurate?
  - Why are Python strings immutable?
  - Why must ‘self’ be used explicitly in method definitions and calls?
  - Why can’t I use an assignment in an expression?
  - Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?
  - Why is join() a string method instead of a list or tuple method?
  - How fast are exceptions?
  - Why isn’t there a switch or case statement in Python?
  - Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?
  - Why can’t lambda expressions contain statements?
  - Can Python be compiled to machine code, C or some other language?
  - How does Python manage memory?
  - Why doesn’t CPython use a more traditional garbage collection scheme?
  - Why isn’t all memory freed when CPython exits?
  - Why are there separate tuple and list data types?
  - How are lists implemented in CPython?
  - How are dictionaries implemented in CPython?
- Why must dictionary keys be immutable?
- Why doesn't `list.sort()` return the sorted list?
- How do you specify and enforce an interface spec in Python?
- Why is there no goto?
- Why can't raw strings (r-strings) end with a backslash?
- Why doesn't Python have a “with” statement for attribute assignments?
- Why are colons required for the if/while/def/class statements?
- Why does Python allow commas at the end of lists and tuples?
Why does Python use indentation for grouping of statements?

Guido van Rossum believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after a while.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```python
if (x <= y)
    x++;
    y--;
z++;
```

Only the `x++` statement is executed if the condition is true, but the indentation leads you to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. If you’re used to reading and writing code that uses one style, you will feel at least slightly uneasy when reading (or being required to write) another style.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python
can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.
Why am I getting strange results with simple arithmetic operations?

See the next question.
Why are floating-point calculations so inaccurate?

Users are often surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It’s not. This has little to do with Python, and much more to do with how the underlying platform handles floating-point numbers.

The float type in CPython uses a C double for storage. A float object’s value is stored in binary floating-point with a fixed precision (typically 53 bits) and Python uses C operations, which in turn rely on the hardware implementation in the processor, to perform floating-point operations. This means that as far as floating-point operations are concerned, Python behaves like many popular languages including C and Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating-point. For example, after:

```
>>> x = 1.2
```

the value stored for x is a (very good) approximation to the decimal value 1.2, but is not exactly equal to it. On a typical machine, the actual stored value is:

```
1.0011001100110011001100110011001100110011001100110011
```
which is exactly:

```
1.19999999999999995559107901499373838305473327636718
```

The typical precision of 53 bits provides Python floats with 15–16
decimal digits of accuracy.

For a fuller explanation, please see the floating point arithmetic
chapter in the Python tutorial.
Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as “elemental” as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string “eight” to anything else.
Why must ‘self’ be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it's more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don’t know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you'd have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>).` This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren’t explicitly declared global), there has to be some way to tell the
interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn’t have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don’t have to search the instance’s directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.
Why can’t I use an assignment in an expression?

Many people used to C or Perl complain that they want to use this C idiom:

```c
while (line = readline(f)) {
    // do something with line
}
```

where in Python you’re forced to write this:

```python
while True:
    line = f.readline()
    if not line:
        break
    ...  # do something with line
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```python
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

The error is a simple typo: \(x = 0\), which assigns 0 to the variable \(x\), was written while the comparison \(x == 0\) is certainly what was intended.
Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

An interesting phenomenon is that most experienced Python programmers recognize the `while True` idiom and don’t seem to be missing the assignment in expression construct much; it’s only newcomers who express a strong desire to add this to the language.

There’s an alternative way of spelling this that seems attractive but is generally less robust than the “while True” solution:

```python
line = f.readline()
while line:
    ... # do something with line...
    line = f.readline()
```

The problem with this is that if you change your mind about exactly how you get the next line (e.g. you want to change it into `sys.stdin.readline()`) you have to remember to change two places in your program – the second occurrence is hidden at the bottom of the loop.

The best approach is to use iterators, making it possible to loop through objects using the `for` statement. For example, file objects support the iterator protocol, so you can write simply:

```python
for line in f:
    ... # do something with line...
```
Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))? 

As Guido said:

(a) For some operations, prefix notation just reads better than postfix – prefix (and infix!) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the easy with which we rewrite a formula like \( x \times (a+b) \) into \( x \times a + x \times b \) to the clumsiness of doing the same thing using a raw OO notation.

(b) When I read code that says \( \text{len}(x) \) I know that it is asking for the length of something. This tells me two things: the result is an integer, and the argument is some kind of container. To the contrary, when I read \( x.\text{len}() \), I have to already know that \( x \) is some kind of container implementing an interface or inheriting from a class that has a standard \( \text{len}() \). Witness the confusion we occasionally have when a class that is not implementing a mapping has a \( \text{get}() \) or \( \text{keys}() \) method, or something that isn’t a file has a \( \text{write}() \) method.

Why is join() a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
"", ".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: “It looks really ugly using a method of a string literal (string constant)”, to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: “I am really telling a sequence to join its members together with a string constant”. Sadly, you aren’t. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split("", ")
```
is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space).

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself. Similar methods exist for bytes and bytearray objects.
How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```python
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn’t the case, you coded it like this:

```python
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

For this specific case, you could also use `value = dict.setdefault(key, getvalue(key))`, but only if the `getvalue()` call is cheap enough because it is evaluated in all cases.
Why isn’t there a switch or case statement in Python?

You can do this easily enough with a sequence of `if... elif... elif... else`. There have been some proposals for switch statement syntax, but there is no consensus (yet) on whether and how to do range tests. See PEP 275 for complete details and the current status.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```python
def function_1(...):
    ...

functions = {'a': function_1,
              'b': function_2,
              'c': self.method_1, ...
}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```python
def visit_a(self, ...):
    ...
    ...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
```
method()

It’s suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.
Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is Stackless Python, which has a completely redesigned interpreter loop that avoids the C stack.
Why can’t lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python's syntactic framework can’t handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you’re too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally-defined function is that you don’t need to invent a name for the function – but that’s just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!
Can Python be compiled to machine code, C or some other language?

Cython compiles a modified version of Python with optional annotations into C extensions. Nuitka is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language. For compiling to Java you can consider VOC.
How does Python manage memory?

The details of Python memory management depend on the implementation. The standard implementation of Python, CPython, uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The gc module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Other implementations (such as Jython or PyPy), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

In some Python implementations, the following code (which is fine in CPython) will probably run out of file descriptors:

```python
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython’s reference counting and destructor scheme, each new assignment to f closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of memory management scheme:
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
Why doesn’t CPython use a more traditional garbage collection scheme?

For one thing, this is not a C standard feature and hence it’s not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for most common platforms, not for all of them, and although it is mostly transparent, it isn’t completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it’s fine to replace the standard malloc() and free() with versions provided by the GC library, an application embedding Python may want to have its own substitute for malloc() and free(), and may not want Python’s. Right now, CPython works with anything that implements malloc() and free() properly.
Why isn’t all memory freed when CPython exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.
Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.
How are lists implemented in CPython?

CPython’s lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array’s length in a list head structure.

This makes indexing a list \( a[i] \) an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don’t require an actual resize.
How are dictionaries implemented in CPython?

CPython’s dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, “Python” could hash to -539294296 while “python”, a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you’re storing keys that all have different hash values, this means that dictionaries take constant time – O(1), in Big-O notation – to retrieve a key.
Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can’t tell that it was being used as a dictionary key, it can’t move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won’t be found because its hash value is different. If you tried to look up the old value it wouldn’t be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn’t work because if you construct a new list with the same value it won’t be found; e.g.:

```python
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

would raise a `KeyError` exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn’t work
because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.

- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.

- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```python
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l) * 555
```

for i, el in enumerate(l):
    try:
        result = result + (hash(el) % 9999999)
    except Exception:
        result = (result % 7777777) + i * 33
return result

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if o1 == o2 (ie o1.__eq__(o2) is True) then hash(o1) == hash(o2) (ie, o1.__hash__() == o2.__hash__()), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of ListWrapper, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.
Why doesn’t list.sort() return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, list.sort() sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won’t be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

If you want to return a new list, use the built-in sorted() function instead. This function creates a new list from a provided iterable, sorts it and returns it. For example, here’s how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):
    ... # do whatever with mydict[key]...
```
How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections.abc` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components. There is also a tool, PyChecker, which can be used to find problems due to subclassing.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple “self test.” Even modules which use complex external interfaces can often be tested in isolation using trivial “stub” emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would.
In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code with an eye to making it easily tested. One increasingly popular technique, test-directed development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.
Why is there no goto?

You can use exceptions to provide a “structured goto” that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the “go” or “goto” constructs of C, Fortran, and other languages. For example:

```python
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass

...  # where to goto
```

This doesn't allow you to jump into the middle of a loop, but that's usually considered an abuse of goto anyway. Use sparingly.
Why can’t raw strings (r-strings) end with a backslash?

More precisely, they can’t end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you’re trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt")  # works fine!
```

If you’re trying to build a pathname for a DOS command, try e.g. one of

```
dir = r"\this\is\my\dos\dir" "\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```
Why doesn’t Python have a “with” statement for attribute assignments?

Python has a ‘with’ statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some language have a construct that looks like this:

```
with obj:
    a = 1  # equivalent to obj.a = 1
    total = total + 1  # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it’s possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler *always* knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```
def foo(a):
    with a:
        print(x)
```

The snippet assumes that “a” must have a member attribute called “x”. However, there is nothing in Python that tells the interpreter this. What
should happen if “a” is, let us say, an integer? If there is a global variable named “x”, will it be used inside the with block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of “with” and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```python
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

write this:

```python
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.
Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```python
if a == b
  print(a)
```  

versus

```python
if a == b:
  print(a)
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it's a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.
Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```python
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it's easier to add more elements because you don't have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```python
x = [
    "fee",
    "fie",
    "foo",
    "fum"
]
```

This list looks like it has four elements, but it actually contains three: “fee”, “fiefoo” and “fum”. Always adding the comma avoids this source
of error.

Allowing the trailing comma may also make programmatic code generation easier.
Library and Extension FAQ

Contents

- Library and Extension FAQ
  - General Library Questions
    - How do I find a module or application to perform task X?
    - Where is the math.py (socket.py, regex.py, etc.) source file?
    - How do I make a Python script executable on Unix?
    - Is there a curses/termcap package for Python?
    - Is there an equivalent to C’s onexit() in Python?
    - Why don’t my signal handlers work?
  - Common tasks
    - How do I test a Python program or component?
    - How do I create documentation from doc strings?
    - How do I get a single keypress at a time?
  - Threads
    - How do I program using threads?
    - None of my threads seem to run: why?
    - How do I parcel out work among a bunch of worker threads?
    - What kinds of global value mutation are thread-safe?
    - Can’t we get rid of the Global Interpreter Lock?
  - Input and Output
    - How do I delete a file? (And other file questions…)
    - How do I copy a file?
    - How do I read (or write) binary data?
    - I can’t seem to use os.read() on a pipe created with os.popen(); why?
    - How do I access the serial (RS232) port?
Why doesn’t closing sys.stdout (stdin, stderr) really close it?

- Network/Internet Programming
  - What WWW tools are there for Python?
  - How can I mimic CGI form submission (METHOD=POST)?
  - What module should I use to help with generating HTML?
  - How do I send mail from a Python script?
  - How do I avoid blocking in the connect() method of a socket?

- Databases
  - Are there any interfaces to database packages in Python?
  - How do you implement persistent objects in Python?

- Mathematics and Numerics
  - How do I generate random numbers in Python?
General Library Questions

How do I find a module or application to perform task X?

Check the Library Reference to see if there’s a relevant standard library module. (Eventually you’ll learn what's in the standard library and will be able to skip this step.)

For third-party packages, search the Python Package Index or try Google or another Web search engine. Searching for “Python” plus a keyword or two for your topic of interest will usually find something helpful.

Where is the math.py (socket.py, regex.py, etc.) source file?

If you can’t find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like mathmodule.c, somewhere in a C source directory (not on the Python Path).

There are (at least) three kinds of modules in Python:

1. modules written in Python (.py);

2. modules written in C and dynamically loaded (.dll, .pyd, .so, .sl, etc);

3. modules written in C and linked with the interpreter; to get a list of these, type:
import sys
print(sys.builtin_module_names)

How do I make a Python script executable on Unix?

You need to do two things: the script file’s mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

```
#!/usr/local/bin/python
```

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the `env` program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's PATH:

```
#!/usr/bin/env python
```

Don’t do this for CGI scripts. The PATH variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.
Occasionally, a user’s environment is so full that the `/usr/bin/env` program fails; or there’s no env program at all. In that case, you can try the following hack (due to Alex Rezinsky):

```bash
#!/bin/sh
"":
exec python $0 ${1+"$@"}
"
```

The minor disadvantage is that this defines the script’s `__doc__` string. However, you can fix that by adding

```python
__doc__ = """...Whatever..."""
```

Is there a curses/termcap package for Python?

For Unix variants: The standard Python source distribution comes with a curses module in the `Modules` subdirectory, though it’s not compiled by default. (Note that this is not available in the Windows distribution – there is no curses module for Windows.)

The `curses` module supports basic curses features as well as many additional functions from ncsurses and SYSV curses such as colour, alternative character set support, pads, and mouse support. This means the module isn’t compatible with operating systems that only have BSD curses, but there don’t seem to be any currently maintained OSes that fall into this category.

For Windows: use the `consolelib` module.

Is there an equivalent to C’s `onexit()` in Python?
The `atexit` module provides a register function that is similar to C's `onexit()`.

**Why don’t my signal handlers work?**

The most common problem is that the signal handler is declared with the wrong argument list. It is called as

```python
handler(signum, frame)
```

so it should be declared with two arguments:

```python
def handler(signum, frame):
    ...
```
Common tasks

How do I test a Python program or component?

Python comes with two testing frameworks. The doctest module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The unittest module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The “global main logic” of your program may be as simple as

```python
if __name__ == "__main__":
    main_logic()
```

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of functions and class behaviours you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it's surprisingly easy. You can make
coding much more pleasant and fun by writing your test functions in parallel with the “production code”, since this makes it easy to find bugs and even design flaws earlier.

“Support modules” that are not intended to be the main module of a program may include a self-test of the module.

```python
if __name__ == '__main__':
    self_test()
```

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using “fake” interfaces implemented in Python.

**How do I create documentation from doc strings?**

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is `epydoc`. `Sphinx` can also include docstring content.

**How do I get a single keypress at a time?**

For Unix variants there are several solutions. It's straightforward to do this using `curses`, but `curses` is a fairly large module to learn.
Threads

How do I program using threads?

Be sure to use the `threading` module and not the `_thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `_thread` module.

Aahz has a set of slides from his threading tutorial that are helpful; see http://www.pythoncraft.com/OSCON2001/.

None of my threads seem to run: why?

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that's long enough for all the threads to finish:

```python
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(T.start(),

time.sleep(10)  # <-------------------------------!
```
But now (on many platforms) the threads don’t run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn’t start a new thread until the previous thread is blocked.

A simple fix is to add a tiny sleep to the start of the run function:

```python
def thread_task(name, n):
    time.sleep(0.001)  # <-------------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(T.start(),

    time.sleep(10)
```

Instead of trying to guess a good delay value for `time.sleep()`, it’s better to use some kind of semaphore mechanism. One idea is to use the `queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

**How do I parcel out work among a bunch of worker threads?**

The easiest way is to use the new `concurrent.futures` module, especially the `ThreadPoolExecutor` class.

Or, if you want fine control over the dispatching algorithm, you can write your own logic manually. Use the `queue` module to create a
queue containing a list of jobs. The Queue class maintains a list of objects and has a .put(obj) method that adds items to the queue and a .get() method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here's a trivial example:

```python
import threading, queue, time

# The worker thread gets jobs off the queue. When this thread assumes there will be no more work and exits. # (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.currentThread(), 'queue empty')
            break
        else:
            print('Worker', threading.currentThread(), 'running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker' + str(i))
    t.start()

# Begin adding work to the queue
```
```python
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

When run, this will produce the following output:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consult the module's documentation for more details; the `Queue` class provides a featureful interface.

**What kinds of global value mutation are thread-safe?**

A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setswitchinterval()`.
Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that “look atomic” really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dict, x, y are objects, i, j are ints):

```python
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

These aren’t:

```python
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects’ `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!
Can't we get rid of the Global Interpreter Lock?

The global interpreter lock (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen recently did a similar experiment in his python-safethread project. Unfortunately, both experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL.

This doesn’t mean that you can’t make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple processes rather than multiple threads. The ProcessPoolExecutor class in the new concurrent.futures module provides an easy way of doing so; the multiprocessing module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as zlib and hashlib already do this.

It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn’t be able to share
objects. Unfortunately, this isn’t likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?
Input and Output

How do I delete a file? (And other file questions…)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don’t exist. `os.removedirs(path)` will remove intermediate directories as long as they’re empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "rb+"), and use `f.truncate(offset); offset defaults to the current seek position. There’s also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where `fd` is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

How do I copy a file?

The `shutil` module contains a `copyfile()` function. Note that on MacOS 9 it doesn’t copy the resource fork and Finder info.

How do I read (or write) binary data?
To read or write complex binary data formats, it's best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```python
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

The ‘>’ in the format string forces big-endian data; the letter ‘h’ reads one “short integer” (2 bytes), and ‘l’ reads one “long integer” (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

**Note:** To read and write binary data, it is mandatory to open the file in binary mode (here, passing "rb" to `open()`). If you use "r" instead (the default), the file will be open in text mode and `f.read()` will return `str` objects rather than `bytes` objects.

I can’t seem to use `os.read()` on a pipe created with `os.popen()`; why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()`
function. Thus, to read \( n \) bytes from a pipe \( p \) created with \texttt{os.popen()}], you need to use \texttt{p.read(n)}.

**How do I access the serial (RS232) port?**

For Win32, POSIX (Linux, BSD, etc.), Jython:

\[
\text{http://pyserial.sourceforge.net}
\]

For Unix, see a Usenet post by Mitch Chapman:

\[
\text{https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com}
\]

**Why doesn’t closing \texttt{sys.stdout (stdin, stderr)} really close it?**

Python file objects are a high-level layer of abstraction on low-level C file descriptors.

For most file objects you create in Python via the built-in \texttt{open()} function, \texttt{f.close()} marks the Python file object as being closed from Python’s point of view, and also arranges to close the underlying C file descriptor. This also happens automatically in \texttt{f}’s destructor, when \texttt{f} becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running \texttt{sys.stdout.close()} marks the Python-level file object as being closed, but does \textit{not} close the associated C file descriptor.

To close the underlying C file descriptor for one of these three, you should first be sure that's what you really want to do (e.g., you may
confuse extension modules trying to do I/O). If it is, use `os.close()`:

```python
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

Or you can use the numeric constants 0, 1 and 2, respectively.
Network/Internet Programming

What WWW tools are there for Python?

See the chapters titled Internet Protocols and Support and Internet Data Handling in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at https://wiki.python.org/moin/WebProgramming.


How can I mimic CGI form submission (METHOD=POST)?

I would like to retrieve web pages that are the result of POSTing a form. Is there existing code that would let me do this easily?

Yes. Here's a simple example that uses urllib.request:

```python
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen(’http://www.some-server.out-there’/cgi-bin/some-cgi-scri
```
with req:
    msg, hdrs = req.read(), req.info()

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.parse.urlencode()`. For example, to send `name=Guy Steele, Jr.`:

```python
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr. '}
                          'name=Guy+Steele%2C+Jr.'
```

**See also:** HOWTO Fetch Internet Resources Using The urllib Package for extensive examples.

What module should I use to help with generating HTML?

You can find a collection of useful links on the Web Programming wiki page.

How do I send mail from a Python script?

Use the standard library module `smtplib`.

Here’s a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```python
import sys, smtplib
fromaddr = input("From: ")
toaddrs = input("To: ").split(', ')
```
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()

A Unix-only alternative uses sendmail. The location of the sendmail program varies between systems; sometimes it is /usr/lib/sendmail, sometimes /usr/sbin/sendmail. The sendmail manual page will help you out. Here's some sample code:

```python
import os

SENDMAIL = ""/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

How do I avoid blocking in the connect() method of a socket?
The `select` module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `.errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the `errno` value. To poll, you can call `connect_ex()` again later – `0` or `errno.EISCONN` indicate that you're connected – or you can pass this socket to `select` to check if it's writable.

**Note:** The `asyncore` module presents a framework-like approach to the problem of writing non-blocking networking code. The third-party `Twisted` library is a popular and feature-rich alternative.
Databases

Are there any interfaces to database packages in Python?

Yes.

Interfaces to disk-based hashes such as DBM and GDBM are also included with standard Python. There is also the sqlite3 module, which provides a lightweight disk-based relational database.

Support for most relational databases is available. See the DatabaseProgramming wiki page for details.

How do you implement persistent objects in Python?

The pickle library module solves this in a very general way (though you still can’t store things like open files, sockets or windows), and the shelve library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects.
Mathematics and Numerics

How do I generate random numbers in Python?

The standard module `random` implements a random number generator. Usage is simple:

```python
import random
random.random()
```

This returns a random floating point number in the range \([0, 1)\).

There are also many other specialized generators in this module, such as:

- `randrange(a, b)` chooses an integer in the range \([a, b)\).
- `uniform(a, b)` chooses a floating point number in the range \([a, b)\).
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Some higher-level functions operate on sequences directly, such as:

- `choice(S)` chooses random element from a given sequence
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly

There's also a `Random` class you can instantiate to create independent multiple random number generators.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
Extending/Embedding FAQ

Contents

- Extending/Embedding FAQ
  - Can I create my own functions in C?
  - Can I create my own functions in C++?
  - Writing C is hard; are there any alternatives?
  - How can I execute arbitrary Python statements from C?
  - How can I evaluate an arbitrary Python expression from C?
  - How do I extract C values from a Python object?
  - How do I use Py_BuildValue() to create a tuple of arbitrary length?
  - How do I call an object's method from C?
  - How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
  - How do I access a module written in Python from C?
  - How do I interface to C++ objects from Python?
  - I added a module using the Setup file and the make fails; why?
  - How do I debug an extension?
  - I want to compile a Python module on my Linux system, but some files are missing. Why?
  - How do I tell “incomplete input” from “invalid input”?
  - How do I find undefined g++ symbols __builtin_new or __pure_virtual?
  - Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?
Can I create my own functions in C?

Yes, you can create built-in modules containing functions, variables, exceptions and even new types in C. This is explained in the document *Extending and Embedding the Python Interpreter*.

Most intermediate or advanced Python books will also cover this topic.
Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.
Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you're trying to do.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.
How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns 0 for success and -1 when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`, see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`. 
How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.
How do I extract C values from a Python object?

That depends on the object’s type. If it’s a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C’s `strlen()` should not be used.

To test the type of an object, first make sure it isn’t `NULL`, and then use `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called ‘abstract’ interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the PyMapping APIs.
How do I use Py_BuildValue() to create a tuple of arbitrary length?

You can't. Use PyTuple_Pack() instead.
How do I call an object’s method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```c
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                   const char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

To call, e.g., a file object’s “seek” method with arguments 10, 0 (assuming the file object pointer is “f”):

```c
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
} else {
    Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` always wants a tuple for the argument list, to call a function without arguments, pass “()” for the format, and to call a function with one argument, surround the argument in parentheses, e.g. “(i)”. 
How do I catch the output from `PyErr_Print()` (or anything that prints to stdout/stderr)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call `print_error`, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `io.StringIO` class:

```python
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

A custom object to do the same would look like this:

```python
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
```
<table>
<thead>
<tr>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello world!</td>
</tr>
</tbody>
</table>
How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```c
module = PyImport_ImportModule("<modulename>";
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<modulename>"]. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```c
attr = PyObject_GetAttrString(module, "<attrname>";
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.
How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the “Extending and Embedding” document. Realize that for the Python run-time system, there isn’t a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see Writing C is hard; are there any alternatives?.

I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)
How do I debug an extension?

When using GDB with dynamically loaded extensions, you can’t set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```
I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python don’t include the /usr/lib/python2.x/config/ directory, which contains various files required for compiling Python extensions.

For Red Hat, install the python-devel RPM to get the necessary files.

For Debian, run `apt-get install python-dev`. 
How do I tell “incomplete input” from “invalid input”?  

Sometimes you want to emulate the Python interactive interpreter’s behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an “if” statement or you didn’t close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the codeop module, which approximates the parser’s behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call PyRun_InteractiveLoop() (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the PyOS_ReadlineFunctionPointer() to point at your custom input function. See Modules/readline.c and Parser/myreadline.c for more hints.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can’t allow the PyRun_InteractiveLoop() to stop while waiting for user input. The one solution then is to call PyParser_ParseString() and test for e.error equal to E_EOF, which means the input is incomplete. Here’s a sample code fragment, untested, inspired by code from Alex Farber:

```c
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
```
```c
#include <compile.h>

int testcomplete(char *code)
    /* code should end in \n */
    /* return -1 for error, 0 for incomplete, 1 for complete */
{
    node *n;
    perrdetail e;

    n = PyParser_ParseString(code, &_PyParser_Grammar,
                              Py_file_input, &e);
    if (n == NULL) {
        if (e.error == E_EOF)
            return 0;
        return -1;
    }

    PyNode_Free(n);
    return 1;
}
```

Another solution is trying to compile the received string with `Py_CompileString()`. If it compiles without errors, try to execute the returned code object by calling `PyEval_EvalCode()`. Otherwise save the input for later. If the compilation fails, find out if it's an error or just more input is required - by extracting the message string from the exception tuple and comparing it to the string “unexpected EOF while parsing”. Here is a complete example using the GNU readline library (you may want to ignore SIGINT while calling readline()):

```c
#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
```
```c
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[]) {
    int i, j, done = 0; /* length of line, code */
    char ps1[] = ">>> ";
    char ps2[] = "... ";
    char *prompt = ps1;
    char *msg, *line, *code = NULL;
    PyObject *src, *glb, *loc;

    Py_Initialize ();
    loc = PyDict_New ();
    glb = PyDict_New ();
    PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

    while (!done) {
        line = readline (prompt);

        if (NULL == line) /* Ctrl-D pressed */
            done = 1;
    } else {
        i = strlen (line);

        if (i > 0) /* save non-empty lines */
            add_history (line);

        if (NULL == code) /* no code */
            j = 0;
    else
            j = strlen (code);
```
code = realloc (code, i + j + 2);
if (NULL == code) /* out of memory */
    exit (1);

if (0 == j) /* code was empty, so */
    code[0] = '\0'; /* keep strncat happy */
strncat (code, line, i); /* append line to code */
code[i + j] = '\n'; /* append '\n' to code */
code[i + j + 1] = '\0';

src = Py_CompileString (code, "<stdin>", Py_single_input);

if (NULL != src) /* compiled just fine - */
{
    if (ps1 == prompt ||
        '\n' == code[i + j - 1]) /* "">
        {
        dum = PyEval_EvalCode (src, glb, loc);
        Py_DECREF (dum);
        Py_XDECREF (src);
        free (code);
        code = NULL;
        if (PyErr_Occurred ())
            PyErr_Print ();
        prompt = ps1;
    }
    /* syntax error or E_EOF? */
else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
    {
    PyErr_Fetch (&exc, &val, &trb); /* clear exception! */

    if (!strcmp (msg, "unexpected EOF while parsing")
    {
    PyErr_Print ();
    Py_DECREF (exc);
    }
Py_XDECREF (val);
Py_XDECREF (trb);
prompt = ps2;
}
else /* some other syntax error */
{
    PyErr_Restore (exc, val, trb);
    PyErr_Print ();
    free (code);
    code = NULL;
    prompt = ps1;
}
}
else /* some non-syntax error */
{
    PyErr_Print ();
    free (code);
    code = NULL;
    prompt = ps1;
}
free (line);
}
}

Py_XDECREF (glb);
Py_XDECREF (loc);
Py_Finalize ();
exit (0);
How do I find undefined g++ symbols __builtin_new or __pure_virtual?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).
Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, http://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).
Python on Windows FAQ

Contents

- Python on Windows FAQ
  - How do I run a Python program under Windows?
  - How do I make Python scripts executable?
  - Why does Python sometimes take so long to start?
  - How do I make an executable from a Python script?
  - Is a *.pyd file the same as a DLL?
  - How can I embed Python into a Windows application?
  - How do I keep editors from inserting tabs into my Python source?
  - How do I check for a keypress without blocking?
How do I run a Python program under Windows?

This is not necessarily a straightforward question. If you are already familiar with running programs from the Windows command line then everything will seem obvious; otherwise, you might need a little more guidance.

Unless you use some sort of integrated development environment, you will end up typing Windows commands into what is variously referred to as a “DOS window” or “Command prompt window”. Usually you can create such a window from your search bar by searching for cmd. You should be able to recognize when you have started such a window because you will see a Windows “command prompt”, which usually looks like this:

```
C:/> 
```

The letter may be different, and there might be other things after it, so you might just as easily see something like:

```
D: \YourName\Projects\Python>
```

depending on how your computer has been set up and what else you have recently done with it. Once you have started such a window, you are well on the way to running Python programs.

You need to realize that your Python scripts have to be processed by another program called the Python interpreter. The interpreter reads your script, compiles it into bytecodes, and then executes the bytecodes to run your program. So, how do you arrange for the interpreter to handle your Python?
First, you need to make sure that your command window recognises the word “py” as an instruction to start the interpreter. If you have opened a command window, you should try entering the command `py` and hitting return:

```
C:\Users\YourName> py
```

You should then see something like:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45)
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

You have started the interpreter in “interactive mode”. That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python's strongest features. Check it by entering a few expressions of your choice and seeing the results:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Many people use the interactive mode as a convenient yet highly programmable calculator. When you want to end your interactive Python session, call the `exit()` function or hold the `ctrl` key down while you enter a `z`, then hit the “Enter” key to get back to your Windows command prompt.

You may also find that you have a Start-menu entry such as Start ▶ Programs ▶ Python 3.x ▶ Python (command line) that results in you
seeing the >>> prompt in a new window. If so, the window will disappear after you call the `exit()` function or enter the `ctrl-z` character; Windows is running a single “python” command in the window, and closes it when you terminate the interpreter.

Now that we know the `py` command is recognized, you can give your Python script to it. You’ll have to give either an absolute or a relative path to the Python script. Let’s say your Python script is located in your desktop and is named `hello.py`, and your command prompt is nicely opened in your home directory so you’re seeing something similar to:

```
C:\Users\YourName>
```

So now you’ll ask the `py` command to give your script to Python by typing `py` followed by your script path:

```
C:\Users\YourName> py Desktop\hello.py
hello
```
How do I make Python scripts executable?

On Windows, the standard Python installer already associates the .py extension with a file type (Python.File) and gives that file type an open command that runs the interpreter (D:\Program Files\Python\python.exe "%1" %*). This is enough to make scripts executable from the command prompt as ‘foo.py’. If you’d rather be able to execute the script by simple typing ‘foo’ with no extension you need to add .py to the PATHEXT environment variable.
Why does Python sometimes take so long to start?

Usually Python starts very quickly on Windows, but occasionally there are bug reports that Python suddenly begins to take a long time to start up. This is made even more puzzling because Python will work fine on other Windows systems which appear to be configured identically.

The problem may be caused by a misconfiguration of virus checking software on the problem machine. Some virus scanners have been known to introduce startup overhead of two orders of magnitude when the scanner is configured to monitor all reads from the filesystem. Try checking the configuration of virus scanning software on your systems to ensure that they are indeed configured identically. McAfee, when configured to scan all file system read activity, is a particular offender.
How do I make an executable from a Python script?

See cx_Freeze for a distutils extension that allows you to create console and GUI executables from Python code. py2exe, the most popular extension for building Python 2.x-based executables, does not yet support Python 3 but a version that does is in development.
Is a *.pyd file the same as a DLL?

Yes, .pyd files are dll's, but there are a few differences. If you have a DLL named foo.pyd, then it must have a function PyInit_foo(). You can then write Python “import foo”, and Python will search for foo.pyd (as well as foo.py, foo.pyc) and if it finds it, will attempt to call PyInit_foo() to initialize it. You do not link your .exe with foo.lib, as that would cause Windows to require the DLL to be present.

Note that the search path for foo.pyd is PYTHONPATH, not the same as the path that Windows uses to search for foo.dll. Also, foo.pyd need not be present to run your program, whereas if you linked your program with a dll, the dll is required. Of course, foo.pyd is required if you want to say import foo. In a DLL, linkage is declared in the source code with __declspec(dllexport). In a .pyd, linkage is defined in a list of available functions.
How can I embed Python into a Windows application?

Embedding the Python interpreter in a Windows app can be summarized as follows:

1. Do _not_ build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to pythonNN.dll; it is typically installed in C:\Windows\System. NN is the Python version, a number such as “33” for Python 3.3.

You can link to Python in two different ways. Load-time linking means linking against pythonNN.lib, while run-time linking means linking against pythonNN.dll. (General note: pythonNN.lib is the so-called “import lib” corresponding to pythonNN.dll. It merely defines symbols for the linker.)

Run-time linking greatly simplifies link options; everything happens at run time. Your code must load pythonNN.dll using the Windows LoadLibraryEx() routine. The code must also use access routines and data in pythonNN.dll (that is, Python's C API's) using pointers obtained by the Windows GetProcAddress() routine. Macros can make using these pointers transparent to any C code that calls routines in Python's C API.

Borland note: convert pythonNN.lib to OMF format using Coff2Omf.exe first.
2. If you use SWIG, it is easy to create a Python “extension module” that will make the app’s data and methods available to Python. SWIG will handle just about all the grungy details for you. The result is C code that you link into your .exe file (!) You do _not_ have to create a DLL file, and this also simplifies linking.

3. SWIG will create an init function (a C function) whose name depends on the name of the extension module. For example, if the name of the module is leo, the init function will be called initleo(). If you use SWIG shadow classes, as you should, the init function will be called initleoc(). This initializes a mostly hidden helper class used by the shadow class.

   The reason you can link the C code in step 2 into your .exe file is that calling the initialization function is equivalent to importing the module into Python! (This is the second key undocumented fact.)

4. In short, you can use the following code to initialize the Python interpreter with your extension module.

   ```c
   #include "python.h"
   ...
   Py_Initialize(); // Initialize Python.
   initmyAppc();   // Initialize (import) the helper
   PyRun_SimpleString("import myApp"); // Import the
   ```

5. There are two problems with Python’s C API which will become apparent if you use a compiler other than MSVC, the compiler used to build pythonNN.dll.

   Problem 1: The so-called “Very High Level” functions that take FILE * arguments will not work in a multi-compiler environment because each compiler’s notion of a struct FILE will be different.
From an implementation standpoint these are very _low_ level functions.

Problem 2: SWIG generates the following code when generating wrappers to void functions:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Alas, Py_None is a macro that expands to a reference to a complex data structure called _Py_NoneStruct inside pythonNN.dll. Again, this code will fail in a multi-compiler environment. Replace such code by:

```
return Py_BuildValue(""");
```

It may be possible to use SWIG’s `%typemap` command to make the change automatically, though I have not been able to get this to work (I’m a complete SWIG newbie).

6. Using a Python shell script to put up a Python interpreter window from inside your Windows app is not a good idea; the resulting window will be independent of your app’s windowing system. Rather, you (or the wxPythonWindow class) should create a “native” interpreter window. It is easy to connect that window to the Python interpreter. You can redirect Python’s i/o to _any_ object that supports read and write, so all you need is a Python object (defined in your extension module) that contains read() and write() methods.
How do I keep editors from inserting tabs into my Python source?

The FAQ does not recommend using tabs, and the Python style guide, **PEP 8**, recommends 4 spaces for distributed Python code; this is also the Emacs python-mode default.

Under any editor, mixing tabs and spaces is a bad idea. MSVC is no different in this respect, and is easily configured to use spaces: Take Tools ▶ Options ▶ Tabs, and for file type “Default” set “Tab size” and “Indent size” to 4, and select the “Insert spaces” radio button.

Python raises `IndentationError` or `TabError` if mixed tabs and spaces are causing problems in leading whitespace. You may also run the `tabnanny` module to check a directory tree in batch mode.
How do I check for a keypress without blocking?

Use the msvcr module. This is a standard Windows-specific extension module. It defines a function `kbhit()` which checks whether a keyboard hit is present, and `getch()` which gets one character without echoing it.
## Contents

- Graphic User Interface FAQ
  - General GUI Questions
  - What platform-independent GUI toolkits exist for Python?
    - Tkinter
    - wxWidgets
    - Qt
    - Gtk+
    - Kivy
    - FLTK
    - OpenGL
  - What platform-specific GUI toolkits exist for Python?
  - Tkinter questions
    - How do I freeze Tkinter applications?
    - Can I have Tk events handled while waiting for I/O?
    - I can’t get key bindings to work in Tkinter: why?
General GUI Questions
What platform-independent GUI toolkits exist for Python?

Depending on what platform(s) you are aiming at, there are several. Some of them haven’t been ported to Python 3 yet. At least Tkinter and Qt are known to be Python 3-compatible.

Tkinter

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called tkinter. This is probably the easiest to install (since it comes included with most binary distributions of Python) and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page. Tcl/Tk is fully portable to the Mac OS X, Windows, and Unix platforms.

wxWidgets

wxWidgets (https://www.wxwidgets.org) is a free, portable GUI class library written in C++ that provides a native look and feel on a number of platforms, with Windows, Mac OS X, GTK, X11, all listed as current stable targets. Language bindings are available for a number of languages including Python, Perl, Ruby, etc.

wxPython is the Python binding for wxwidgets. While it often lags slightly behind the official wxWidgets releases, it also offers a number of features via pure Python extensions that are not available in other language bindings. There is an active wxPython user and developer community.

Both wxWidgets and wxPython are free, open source, software with
permissive licences that allow their use in commercial products as well as in freeware or shareware.

**Qt**

There are bindings available for the Qt toolkit (using either PyQt or PySide) and for KDE (PyKDE4). PyQt is currently more mature than PySide, but you must buy a PyQt license from Riverbank Computing if you want to write proprietary applications. PySide is free for all applications.

Qt 4.5 upwards is licensed under the LGPL license; also, commercial licenses are available from The Qt Company.

**Gtk+**

The GObject introspection bindings for Python allow you to write GTK+ 3 applications. There is also a Python GTK+ 3 Tutorial.

The older PyGtk bindings for the Gtk+ 2 toolkit have been implemented by James Henstridge; see <http://www.pygtk.org>.

**Kivy**

Kivy is a cross-platform GUI library supporting both desktop operating systems (Windows, macOS, Linux) and mobile devices (Android, iOS). It is written in Python and Cython, and can use a range of windowing backends.

Kivy is free and open source software distributed under the MIT license.
**FLTK**

Python bindings for the FLTK toolkit, a simple yet powerful and mature cross-platform windowing system, are available from the PyFLTK project.

**OpenGL**

For OpenGL bindings, see PyOpenGL.
What platform-specific GUI toolkits exist for Python?

By installing the PyObjc Objective-C bridge, Python programs can use Mac OS X's Cocoa libraries.

Pythonwin by Mark Hammond includes an interface to the Microsoft Foundation Classes and a Python programming environment that's written mostly in Python using the MFC classes.
Tkinter questions

How do I freeze Tkinter applications?

Freeze is a tool to create stand-alone applications. When freezing Tkinter applications, the applications will not be truly stand-alone, as the application will still need the Tcl and Tk libraries.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the TCL_LIBRARY and TK_LIBRARY environment variables.

To get truly stand-alone applications, the Tcl scripts that form the library have to be integrated into the application as well. One tool supporting that is SAM (stand-alone modules), which is part of the Tix distribution (http://tix.sourceforge.net/).

Build Tix with SAM enabled, perform the appropriate call to Tclsam_init(), etc. inside Python's Modules/tkappinit.c, and link with libtclsam and libtksam (you might include the Tix libraries as well).

Can I have Tk events handled while waiting for I/O?

On platforms other than Windows, yes, and you don't even need threads! But you'll have to restructure your I/O code a bit. Tk has the equivalent of Xt's XtAddInput() call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See File Handlers.
I can’t get key bindings to work in Tkinter: why?

An often-heard complaint is that event handlers bound to events with the `bind()` method don’t get handled even when the appropriate key is pressed.

The most common cause is that the widget to which the binding applies doesn’t have “keyboard focus”. Check out the Tk documentation for the focus command. Usually a widget is given the keyboard focus by clicking in it (but not for labels; see the `takefocus` option).
“Why is Python Installed on my Computer?” FAQ
What is Python?

Python is a programming language. It’s used for many different applications. It’s used in some high schools and colleges as an introductory programming language because Python is easy to learn, but it’s also used by professional software developers at places such as Google, NASA, and Lucasfilm Ltd.

If you wish to learn more about Python, start with the Beginner’s Guide to Python.
Why is Python installed on my machine?

If you find Python installed on your system but don’t remember installing it, there are several possible ways it could have gotten there.

- Perhaps another user on the computer wanted to learn programming and installed it; you'll have to figure out who’s been using the machine and might have installed it.
- A third-party application installed on the machine might have been written in Python and included a Python installation. There are many such applications, from GUI programs to network servers and administrative scripts.
- Some Windows machines also have Python installed. At this writing we’re aware of computers from Hewlett-Packard and Compaq that include Python. Apparently some of HP/Compaq's administrative tools are written in Python.
- Many Unix-compatible operating systems, such as Mac OS X and some Linux distributions, have Python installed by default; it’s included in the base installation.
Can I delete Python?

That depends on where Python came from.

If someone installed it deliberately, you can remove it without hurting anything. On Windows, use the Add/Remove Programs icon in the Control Panel.

If Python was installed by a third-party application, you can also remove it, but that application will no longer work. You should use that application's uninstaller rather than removing Python directly.

If Python came with your operating system, removing it is not recommended. If you remove it, whatever tools were written in Python will no longer run, and some of them might be important to you. Reinstalling the whole system would then be required to fix things again.
Glossary

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3
A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3 - Automated Python 2 to 3 code translation.

abstract base class
Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don’t inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the abc module documentation. Python comes with many built-in ABCs for data structures (in the collections.abc module), numbers (in the numbers module), streams (in the io module), import finders and loaders (in the importlib.abc module). You can create your own ABCs with
the abc module.

**annotation**

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a type hint.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See variable annotation, function annotation, PEP 484 and PEP 526, which describe this functionality.

**argument**

A value passed to a function (or method) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, 3 and 5 are both keyword arguments in the following calls to `complex()`:

  ```python
  complex(real=3, imag=5)
  complex(**{'real': 3, 'imag': 5})
  ```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, 3 and 5 are both positional arguments in the following calls:

  ```python
   complex(3, 5)
  complex(*(3, 5))
  ```
Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry, the FAQ question on the difference between arguments and parameters, and PEP 362.

**asynchronous context manager**
An object which controls the environment seen in an async with statement by defining \_aenter\_() and \_aexit\_() methods. Introduced by PEP 492.

**asynchronous generator**
A function which returns an asynchronous generator iterator. It looks like a coroutine function defined with async def except that it contains yield expressions for producing a series of values usable in an async for loop.

Usually refers to an asynchronous generator function, but may refer to an asynchronous generator iterator in some contexts. In cases where the intended meaning isn’t clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain await expressions as well as async for, and async with statements.

**asynchronous generator iterator**
An object created by a asynchronous generator function.

This is an asynchronous iterator which when called using the \_anext\_() method returns an awaitable object which will
execute the body of the asynchronous generator function until the next yield expression.

Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the asynchronous generator iterator effectively resumes with another awaitable returned by __anext__(), it picks up where it left off. See PEP 492 and PEP 525.

**asynchronous iterable**
An object, that can be used in an async for statement. Must return an asynchronous iterator from its __aiter__() method. Introduced by PEP 492.

**asynchronous iterator**
An object that implements the __aiter__() and __anext__() methods. __anext__ must return an awaitable object. async for resolves the awaitables returned by an asynchronous iterator's __anext__() method until it raises a StopAsyncIteration exception. Introduced by PEP 492.

**attribute**
A value associated with an object which is referenced by name using dotted expressions. For example, if an object o has an attribute a it would be referenced as o.a.

**awaitable**
An object that can be used in an await expression. Can be a coroutine or an object with an __await__() method. See also PEP 492.

**BDFL**
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python’s creator.
**binary file**

A file object able to read and write bytes-like objects. Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), sys.stdin.buffer, sys.stdout.buffer, and instances of io.BytesIO and gzip.GzipFile.

See also text file for a file object able to read and write str objects.

**bytes-like object**

An object that supports the Buffer Protocol and can export a C-contiguous buffer. This includes all bytes, bytearray, and array.array objects, as well as many common memoryview objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as “read-write bytes-like objects”. Example mutable buffer objects include bytearray and a memoryview of a bytearray. Other operations require the binary data to be stored in immutable objects (“read-only bytes-like objects”); examples of these include bytes and a memoryview of a bytes object.

**bytecode**

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in .pyc files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code
corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

**class**
A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**class variable**
A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

**coercion**
The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

**complex number**
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written i in mathematics or j in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a j suffix, e.g., `3+1j`. To get access to complex
equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

**context manager**
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See PEP 343.

**contiguous**
A buffer is considered contiguous exactly if it is either `C-contiguous` or `Fortran contiguous`. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

**coroutine**
Coroutines is a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also PEP 492.

**coroutine function**
A function which returns a `coroutine` object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by PEP 492.

**CPython**
The canonical implementation of the Python programming
language, as distributed on python.org. The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.

**decorator**

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```python
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

**descriptor**

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super
classes.

For more information about descriptors’ methods, see Implementing Descriptors.

dictionary
An associative array, where arbitrary keys are mapped to values. The keys can be any object with __hash__() and __eq__() methods. Called a hash in Perl.

dictionary view
The objects returned from dict.keys(), dict.values(), and dict.items() are called dictionary views. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use list(dictview). See Dictionary view objects.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the __doc__ attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing
A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used (“If it looks like a duck and quacks like a duck, it must be a duck.”) By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically
employs `hasattr()` tests or **EAFP** programming.

**EAFP**
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the **LBYL** style common to many other languages such as C.

**expression**
A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also **statements** which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

**extension module**
A module written in C or C++, using Python’s C API to interact with the core and with user code.

**f-string**
String literals prefixed with `'f'` or `'F'` are commonly called “f-strings” which is short for formatted string literals. See also PEP 498.

**file object**
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called file-like objects or streams.
There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

**file-like object**
A synonym for file object.

**finder**
An object that tries to find the loader for a module that is being imported.

Since Python 3.3, there are two types of finder: meta path finders for use with `sys.meta_path`, and path entry finders for use with `sys.path_hooks`.

See PEP 302, PEP 420 and PEP 451 for much more detail.

**floor division**
Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to 2 in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded downward. See PEP 238.

**function**
A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the Function definitions section.

**function annotation**
An annotation of a function parameter or return value.

Function annotations are usually used for type hints: for example,
this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```python
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Function annotation syntax is explained in section Function definitions.

See variable annotation and PEP 484, which describe this functionality.

```
>>> import __future__
>>> __future__.division
Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection
The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the gc module.

generator
A function which returns a **generator iterator**. It looks like a normal function except that it contains **yield** expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn’t clear, using the full terms avoids ambiguity.

**generator iterator**

An object created by a **generator** function.

Each **yield** temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the **generator iterator** resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**generator expression**

An expression that returns an iterator. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```python
>>> sum(i*i for i in range(10))  # sum of squares 0, 1, 4, ..., 81
285
```

**generic function**

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.
See also the single dispatch glossary entry, the `functools.singledispatch()` decorator, and PEP 443.

**GIL**

See global interpreter lock.

**global interpreter lock**

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

**hash-based pyc**

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See Cached bytecode invalidation.

**hashable**

An object is *hashable* if it has a hash value which never changes
during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

**IDLE**
An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

**immutable**
An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

**import path**
A list of locations (or path entries) that are searched by the path based finder for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

**importing**
The process by which Python code in one module is made
available to Python code in another module.

**importer**

An object that both finds and loads a module; both a *finder* and *loader* object.

**interactive**

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch python with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

**interpreted**

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also **interactive**.

**interpreter shutdown**

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the **__main__**
module or the script being run has finished executing.

**iterable**

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, `file objects`, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements `Sequence` semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also `iterator`, `sequence`, and `generator`.

**iterator**

An object representing a stream of data. Repeated calls to the iterator’s `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes.
A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types.

**key function**
A key function or collation function is a callable that returns a value used for sorting or ordering. For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include min(), max(), sorted(), list.sort(), heapq.merge(), heapq.nsmallest(), heapq.nlargest(), and itertools.groupby().

There are several ways to create a key function. For example, the str.lower() method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a lambda expression such as lambda r: (r[0], r[2]). Also, the operator module provides three key function constructors: attrgetter(), itemgetter(), and methodcaller(). See the Sorting HOW TO for examples of how to create and use key functions.

**keyword argument**
See argument.

**lambda**
An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

**LBYL**

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes `key` from `mapping` after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

**list**

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

**list comprehension**

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

**loader**

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a finder. See PEP 302 for details and `importlib.abc.Loader` for an abstract
**base class.**

**mapping**
A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

**meta path finder**
A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from path entry finders.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

**metaclass**
The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in Metaclasses.

**method**
A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`).
See function and nested scope.

**method resolution order**
Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

**module**
An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

See also package.

**module spec**
A namespace containing the import-related information used to load a module. An instance of importlib.machinery.ModuleSpec.

**MRO**
See method resolution order.

**mutable**
Mutable objects can change their value but keep their id(). See also immutable.

**named tuple**
Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the year is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the
factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

**namespace**

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**namespace package**

A [PEP 420](https://www.python.org/dev/peps/pep-0420/) package which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a regular package because they have no `__init__.py` file.

See also module.

**nested scope**

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.
new-style class
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python’s newer, versatile features like __slots__, descriptors, properties, __getattribute__(), class methods, and static methods.

object
Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.

package
A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an __path__ attribute.

See also regular package and namespace package.

parameter
A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- **positional-or-keyword**: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example foo and bar in the following:

  ```python
def func(foo, bar=\texttt{None}): \ldots
  ```

- **positional-only**: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. abs()).

- **keyword-only**: specifies an argument that can be supplied only
by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare * in the parameter list of the function definition before them, for example kw_only1 and kw_only2 in the following:

```python
def func(arg, *, kw_only1, kw_only2): ...
```

- **var-positional**: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *, for example args in the following:

```python
def func(*args, **kwargs): ...
```

- **var-keyword**: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with **, for example kwargs in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, the inspect.Parameter class, the Function definitions section, and PEP 362.

**path entry**

A single location on the import path which the path based finder consults to find modules for importing.

**path entry finder**
A **finder** returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

**path entry hook**

A callable on the `sys.path_hook` list which returns a path entry finder if it knows how to find modules on a specific path entry.

**path based finder**

One of the default *meta path finders* which searches an import path for modules.

**path-like object**

An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by PEP 519.

**PEP**

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into
Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

**portion**
A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.

**positional argument**
See **argument**.

**provisional API**
A provisional API is one which has been deliberately excluded from the standard library’s backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a “solution of last resort” - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.

**provisional package**
See **provisional API**.

**Python 3000**
Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

**Pythonic**

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```python
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```python
for piece in food:
    print(piece)
```

**qualified name**

A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in [PEP 3155](https://www.python.org/dev/peps/pep-03155/). For top-level functions and classes, the qualified name is the same as the object's name:

```python
>>> class C:
    ...
    class D:
    ...    def meth(self):
    ...        pass
    ...
>>> C.__qualname__
'C'
```
When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```
An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

**single dispatch**
A form of generic function dispatch where the implementation is chosen based on the type of a single argument.

**slice**
An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses slice objects internally.

**special method**
A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in Special method names.

**statement**
A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword,
such as `if`, `while` or `for`.

**struct sequence**
A tuple with named elements. Struct sequences expose an interface similar to `named tuple` in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

**text encoding**
A codec which encodes Unicode strings to bytes.

**text file**
A file object able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also binary file for a file object able to read and write bytes-like objects.

**triple-quoted string**
A string which is bound by three instances of either a quotation mark (`'`) or an apostrophe (`). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

**type**
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its
__class__ attribute or can be retrieved with `type(obj)`.

**type alias**
A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying type hints. For example:

```python
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List
    pass
```

could be made more readable like this:

```python
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List
    pass
```

See `typing` and **PEP 484**, which describe this functionality.

**type hint**
An annotation that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using
typing.get_type_hints().

See typing and PEP 484, which describe this functionality.

universal newlines
A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention \n, the Windows convention \r\n, and the old Macintosh convention \r. See PEP 278 and PEP 3116, as well as bytes.splitlines() for an additional use.

variable annotation
An annotation of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```python
class C:
    field: 'annotation'
```

Variable annotations are usually used for type hints: for example this variable is expected to take int values:

```python
count: int = 0
```

Variable annotation syntax is explained in section Annotated assignment statements.

See function annotation, PEP 484 and PEP 526, which describe this functionality.

virtual environment
A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution
packages without interfering with the behaviour of other Python applications running on the same system.

See also venv.

**virtual machine**
A computer defined entirely in software. Python’s virtual machine executes the bytecode emitted by the bytecode compiler.

**Zen of Python**
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “import this” at the interactive prompt.
About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the Dealing with Bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.
Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!
Dealing with Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python.

It can be sometimes faster to fix bugs yourself and contribute patches to Python as it streamlines the process and involves less people. Learn how to contribute.
Documentation bugs

If you find a bug in this documentation or would like to propose an improvement, please submit a bug report on the tracker. If you have a suggestion on how to fix it, include that as well.

If you're short on time, you can also email documentation bug reports to docs@python.org (behavioral bugs can be sent to python-list@python.org). ‘docs@’ is a mailing list run by volunteers; your request will be noticed, though it may take a while to be processed.

See also: Documentation bugs on the Python issue tracker
Using the Python issue tracker

Bug reports for Python itself should be submitted via the Python Bug Tracker (https://bugs.python.org/). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the top of the page.

If the problem you’re reporting is not already in the bug tracker, go back to the Python Bug Tracker and log in. If you don’t already have a tracker account, select the “Register” link or, if you use OpenID, one of the OpenID provider logos in the sidebar. It is not possible to submit a bug report anonymously.

Being now logged in, you can submit a bug. Select the “Create New” link in the sidebar to open the bug reporting form.

The submission form has a number of fields. For the “Title” field, enter a very short description of the problem; less than ten words is good. In the “Type” field, select the type of your problem; also select the “Component” and “Versions” to which the bug relates.

In the “Comment” field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include whether any extension modules were involved, and what hardware
and software platform you were using (including version information as appropriate).

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See also:

**How to Report Bugs Effectively**

Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

**Bug Writing Guidelines**

Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
Getting started contributing to Python yourself

Beyond just reporting bugs that you find, you are also welcome to submit patches to fix them. You can find more information on how to get started patching Python in the Python Developer’s Guide. If you have questions, the core-mentorship mailing list is a friendly place to get answers to any and all questions pertaining to the process of fixing issues in Python.
Copyright

Python and this documentation is:

Copyright © 2001-2019 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See History and License for complete license and permissions information.
History and License
History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python’s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru</td>
<td></td>
<td>1995-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>Previous-Version</td>
<td>Year</td>
<td>Organization</td>
<td>GPL-Compatible</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------</td>
<td>------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>1.5.2</td>
<td>1.2</td>
<td>1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2 and above</td>
<td>2.1.1</td>
<td>2001-now</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** GPL-compatible doesn’t mean that we’re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don’t.

Thanks to the many outside volunteers who have worked under Guido’s direction to make these releases possible.
Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 3.7.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.7.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.7.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All Rights Reserved" are retained in Python 3.7.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.7.3 or any part thereof, and derivative work available to others as provided herein, then Licensee agrees to include in any such work a brief summary of the changes made to Python 3.7.3.

4. PSF is making Python 3.7.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.7.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER PARTY FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES FOR ANY MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.3.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade name in a trademark sense to endorse or promote products of third party.

8. By copying, installing or otherwise using Python, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software")

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display derivative works, distribute, and otherwise use the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USER FOR ANY INCIDENTAL,
SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF.

5. This License Agreement will automatically terminate upon material breach of
its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
by the law of the State of California, excluding any conflict of laws provisions.
Nothing in this License Agreement shall be deemed to create any relationship
of agency, partnership, or joint venture between BeOpen and Licensee. This License
Agreement does not grant permission to use BeOpen trademarks in trademark sense to endorse or promote products of
third party. As an exception, the "BeOpen Python" logos available at
http://www.pythonlabs.com/logos.html may be used granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research
Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
("CNRI"), and the Individual or Organization ("Licensee") accessing and
otherwise using Python 1.6.1 software in source or binary form and its
associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI grants Licensee a nonexclusive, royalty-free, world
analyze, test, perform and/or display publicly, provide, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting quotes) is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions.

Notwithstanding the foregoing, with regard to derivative works based on Python
1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only with respect to Paragraphs 4, 5, and 7 of this License Agreement shall be deemed to create no partnership, or joint venture between CNRI and Licensee. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee.

8. By clicking on the "ACCEPT" button where indicated or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum, Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS SOFTWARE.
Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Mersenne Twister

The _random module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_generator or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.  
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

Sockets

The socket module uses the functions, getaddrinfo(), and getnameinfo(), which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of the authors and contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written agreement.
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULT OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Cookie management

The `http.cookies` module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE PERFORMANCE OF THIS SOFTWARE.
Execution tracing

The `trace` module contains the following notice:

```plaintext
portions copyright 2001, Autonomous Zones Industries
err... reserved and offered to the public under the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum,

Permission to use, copy, modify, and distribute this
its associated documentation for any purpose without
granted, provided that the above copyright notice appears
and that both that copyright notice and this permission
supporting documentation, and that the name of neither
Bioreason or Mojam Media be used in advertising or pro-
distribution of the software without specific, writ
```
The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

XML Remote Procedure Calls

The xmlrpc.client module contains the following notice:

```
The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh
```
By obtaining, using, and/or copying this software and its associated documentation, you agree that you have read and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice and all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

---

test_epoll

The test_epoll module contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation ("Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above
   notice, this list of conditions and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above
   notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
```

SipHash24

The file Python/pyhash.c contains Marek Majkowski’ implementation of Dan Bernstein’s SipHash24 algorithm. The contains the following note:

<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any of this software and associated documentation files in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

</MIT License>

Original location:
https://github.com/majek/csiphash/

Solution inspired by code from:
Samuel Neves (supercop/crypto_auth/siphash24/little)
dj (supercop/crypto_auth/siphash24/little2)
Jean-Philippe Aumasson (https://131002.net/siphash)
strtod and dtoa

The file Python/dtoa.c, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```c
/*****************************/
/*
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which includes or modification of this software and in all copies of the supporting documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
="/*****************************/
```

OpenSSL

The modules hashlib, posix, ssl, crypt use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the
LICENSE ISSUES
===============

The OpenSSL toolkit stays under a dual license, i.e. the OpenSSL License and the original SSLeay license. See below for the actual license texts. Actually both are Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License
-----------------

/* ==----------------------------------------------------------------------
 * Copyright (c) 1998-2008 The OpenSSL Project.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed for use in the OpenSSL
 * Toolkit. (http://www.openssl.org)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without prior written permission. For written permi
openssl-core@openssl.org.

5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without the prior written permission of the OpenSSL Project.

6. Redistributions of any form whatsoever must acknowledge:
   "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL Project AS IS AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL Project OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

====================================================================

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).
This product includes software written by Tim Hudson (tjh@cryptsoft.com).

*/

Original SSLeay License
-----------------------

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com).
 * All rights reserved.
 */
This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are aheared to. The following conditions apply to all code found in this distribution, lhash, DES, etc., code; not just the SSL code. included with this distribution is covered by except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed.

If this package is used in a product, Eric Young as the author of the parts of the library used This can be in the form of a textual message or in documentation (online or textual) provided

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"
   The word 'cryptographic' can be left out if being used are not cryptographic related :-(
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:
   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, 
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS 
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR, 
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, 
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT 
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR 
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, 
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHER 
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED 
* SUCH DAMAGE.
*
* The licence and distribution terms for any public 
* derivative of this code cannot be changed. i.e. 
* copied and put under another distribution licence 
* [including the GNU Public Licence.]
*/

expat

The pyexpat extension is built using an included copy of the expat 
sources unless the build is configured --with-system-expat:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software 
and Clark Cooper

Permission is hereby granted, free of charge, to any 
who copies and distributes the Software in source and/or 
binary forms, to deal in the Software without restriction, 
including without limitation the rights to use, copy, modify, 
distribute, sublicense, and/or sell copies of the Software, 
and to permit persons to whom the Software is furnished to 
the following conditions:

The above copyright notice and this permission notice 
in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
zlib

The zlib extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and not misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly Mark Adler
jlush@gzip.org madler@alumni.caltech.edu

cfuhash

The implementation of the hash table used by the tracemalloc is
based on the cfuhash project:

Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
libmpdec

The _decimal module is built using an included copy of the libmpdec library unless the build is configured `-with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```
Index

Index pages by letter:

Symbols | _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Full index on one page (can be huge)
What’s New in Python

The “What’s New in Python” series of essays takes tours through the most important changes between major Python versions. They are a “must read” for anyone wishing to stay up-to-date after a new release.

- **What’s New In Python 3.7**
  - Summary – Release Highlights
  - New Features
  - Other Language Changes
  - New Modules
  - Improved Modules
  - C API Changes
  - Build Changes
  - Optimizations
  - Other CPython Implementation Changes
  - Deprecated Python Behavior
  - Deprecated Python modules, functions and methods
  - Deprecated functions and types of the C API
  - Platform Support Removals
  - API and Feature Removals
  - Module Removals
  - Windows-only Changes
  - Porting to Python 3.7
  - Notable changes in Python 3.7.1
  - Notable changes in Python 3.7.2

- **What’s New In Python 3.6**
  - Summary – Release highlights
  - New Features
  - Other Language Changes
  - New Modules
  - Improved Modules
  - Optimizations
• Build and C API Changes
• Other Improvements
• Deprecated
• Removed
• Porting to Python 3.6
• Notable changes in Python 3.6.2
• Notable changes in Python 3.6.4
• Notable changes in Python 3.6.5
• Notable changes in Python 3.6.7

• What’s New In Python 3.5
  • Summary – Release highlights
  • New Features
  • Other Language Changes
  • New Modules
  • Improved Modules
  • Other module-level changes
  • Optimizations
  • Build and C API Changes
  • Deprecated
  • Removed
  • Porting to Python 3.5
  • Notable changes in Python 3.5.4

• What’s New In Python 3.4
  • Summary – Release Highlights
  • New Features
  • New Modules
  • Improved Modules
  • CPython Implementation Changes
  • Deprecated
  • Removed
  • Porting to Python 3.4
  • Changed in 3.4.3

• What’s New In Python 3.3
  • Summary – Release highlights
- PEP 405: Virtual Environments
- PEP 420: Implicit Namespace Packages
- PEP 3118: New memoryview implementation and buffer protocol documentation
- PEP 393: Flexible String Representation
- PEP 397: Python Launcher for Windows
- PEP 3151: Reworking the OS and IO exception hierarchy
- PEP 380: Syntax for Delegating to a Subgenerator
- PEP 409: Suppressing exception context
- PEP 414: Explicit Unicode literals
- PEP 3155: Qualified name for classes and functions
- PEP 412: Key-Sharing Dictionary
- PEP 362: Function Signature Object
- PEP 421: Adding sys.implementation
- Using importlib as the Implementation of Import

Other Language Changes

- A Finer-Grained Import Lock
- Builtin functions and types
- New Modules
- Improved Modules
- Optimizations
- Build and C API Changes
- Deprecated
- Porting to Python 3.3

- What’s New In Python 3.2
  - PEP 384: Defining a Stable ABI
  - PEP 389: Argparse Command Line Parsing Module
  - PEP 391: Dictionary Based Configuration for Logging
  - PEP 3148: The concurrent.futures module
  - PEP 3147: PYC Repository Directories
  - PEP 3149: ABI Version Tagged .so Files
  - PEP 3333: Python Web Server Gateway Interface v1.0.1
  - Other Language Changes
  - New, Improved, and Deprecated Modules
- Multi-threading
- Optimizations
- Unicode
- Codecs
- Documentation
- IDLE
- Code Repository
- Build and C API Changes
- Porting to Python 3.2

- **What’s New In Python 3.1**
  - PEP 372: Ordered Dictionaries
  - PEP 378: Format Specifier for Thousands Separator
  - Other Language Changes
  - New, Improved, and Deprecated Modules
  - Optimizations
  - IDLE
  - Build and C API Changes
  - Porting to Python 3.1

- **What’s New In Python 3.0**
  - Common Stumbling Blocks
  - Overview Of Syntax Changes
  - Changes Already Present In Python 2.6
  - Library Changes
  - **PEP 3101**: A New Approach To String Formatting
  - Changes To Exceptions
  - Miscellaneous Other Changes
  - Build and C API Changes
  - Performance
  - Porting To Python 3.0

- **What’s New in Python 2.7**
  - The Future for Python 2.x
  - Changes to the Handling of Deprecation Warnings
  - Python 3.1 Features
  - PEP 372: Adding an Ordered Dictionary to collections
• PEP 378: FormatSpecifier for Thousands Separator
• PEP 389: The argparse Module for Parsing Command Lines
• PEP 391: Dictionary-Based Configuration For Logging
• PEP 3106: Dictionary Views
• PEP 3137: The memoryview Object
• Other Language Changes
• New and Improved Modules
• Build and C API Changes
• Other Changes and Fixes
• Porting to Python 2.7
• New Features Added to Python 2.7 Maintenance Releases
• Acknowledgements

What's New in Python 2.6

• Python 3.0
• Changes to the Development Process
• PEP 343: The ‘with’ statement
• PEP 366: Explicit Relative Imports From a Main Module
• PEP 370: Per-user site-packages Directory
• PEP 371: The multiprocessing Package
• PEP 3101: Advanced String Formatting
• PEP 3105: print As a Function
• PEP 3110: Exception-Handling Changes
• PEP 3112: Byte Literals
• PEP 3116: New I/O Library
• PEP 3118: Revised Buffer Protocol
• PEP 3119: Abstract Base Classes
• PEP 3127: Integer Literal Support and Syntax
• PEP 3129: Class Decorators
• PEP 3141: A Type Hierarchy for Numbers
• Other Language Changes
• New and Improved Modules
• Deprecations and Removals
• Build and C API Changes
• Porting to Python 2.6
• What’s New in Python 2.5
  • PEP 308: Conditional Expressions
  • PEP 309: Partial Function Application
  • PEP 314: Metadata for Python Software Packages v1.1
  • PEP 328: Absolute and Relative Imports
  • PEP 338: Executing Modules as Scripts
  • PEP 341: Unified try/except/finally
  • PEP 342: New Generator Features
  • PEP 343: The ‘with’ statement
  • PEP 352: Exceptions as New-Style Classes
  • PEP 353: Using ssize_t as the index type
  • PEP 357: The ‘__index__’ method

• Other Language Changes
  • New, Improved, and Removed Modules
  • Build and C API Changes
  • Porting to Python 2.5

• What’s New in Python 2.4
  • PEP 218: Built-In Set Objects
  • PEP 237: Unifying Long Integers and Integers
  • PEP 289: Generator Expressions
  • PEP 292: Simpler String Substitutions
  • PEP 318: Decorators for Functions and Methods
  • PEP 322: Reverse Iteration
  • PEP 324: New subprocess Module
  • PEP 327: Decimal Data Type
  • PEP 328: Multi-line Imports
  • PEP 331: Locale-Independent Float/String Conversions

• Other Language Changes
  • New, Improved, and Deprecated Modules
  • Build and C API Changes
  • Porting to Python 2.4

• Acknowledgements
• What's New in Python 2.3
  ○ PEP 218: A Standard Set Datatype
  ○ PEP 255: Simple Generators
  ○ PEP 263: Source Code Encodings
  ○ PEP 273: Importing Modules from ZIP Archives
  ○ PEP 277: Unicode file name support for Windows NT
  ○ PEP 278: Universal Newline Support
  ○ PEP 279: enumerate()
  ○ PEP 282: The logging Package
  ○ PEP 285: A Boolean Type
  ○ PEP 293: Codec Error Handling Callbacks
  ○ PEP 301: Package Index and Metadata for Distutils
  ○ PEP 302: New Import Hooks
  ○ PEP 305: Comma-separated Files
  ○ PEP 307: Pickle Enhancements
  ○ Extended Slices
  ○ Other Language Changes
  ○ New, Improved, and Deprecated Modules
  ○ Pymalloc: A Specialized Object Allocator
  ○ Build and C API Changes
  ○ Other Changes and Fixes
  ○ Porting to Python 2.3
  ○ Acknowledgements
• What's New in Python 2.2
  ○ Introduction
  ○ PEPs 252 and 253: Type and Class Changes
  ○ PEP 234: Iterators
  ○ PEP 255: Simple Generators
  ○ PEP 237: Unifying Long Integers and Integers
  ○ PEP 238: Changing the Division Operator
  ○ Unicode Changes
  ○ PEP 227: Nested Scopes
  ○ New and Improved Modules
  ○ Interpreter Changes and Fixes
- Other Changes and Fixes
- Acknowledgements

### What's New in Python 2.1
- Introduction
- PEP 227: Nested Scopes
- PEP 236: `__future__` Directives
- PEP 207: Rich Comparisons
- PEP 230: Warning Framework
- PEP 229: New Build System
- PEP 205: Weak References
- PEP 232: Function Attributes
- PEP 235: Importing Modules on Case-Insensitive Platforms
- PEP 217: Interactive Display Hook
- PEP 208: New Coercion Model
- PEP 241: Metadata in Python Packages
- New and Improved Modules
- Other Changes and Fixes
- Acknowledgements

### What's New in Python 2.0
- Introduction
- What About Python 1.6?
- New Development Process
- Unicode
- List Comprehensions
- Augmented Assignment
- String Methods
- Garbage Collection of Cycles
- Other Core Changes
- Porting to 2.0
- Extending/Embedding Changes
- Distutils: Making Modules Easy to Install
- XML Modules
- Module changes
- New modules
- IDLE Improvements
- Deleted and Deprecated Modules
- Acknowledgements

The “Changelog” is an HTML version of the file built from the contents of the Misc/NEWS.d directory tree, which contains all nontrivial changes to Python for the current version.

- Changelog
The Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.
This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python’s most noteworthy features, and will give you a good idea of the language’s flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in *The Python Standard Library*.

The **Glossary** is also worth going through.

- **1. Whetting Your Appetite**
- **2. Using the Python Interpreter**
  - 2.1. Invoking the Interpreter
    - 2.1.1. Argument Passing
    - 2.1.2. Interactive Mode
  - 2.2. The Interpreter and Its Environment
    - 2.2.1. Source Code Encoding
- **3. An Informal Introduction to Python**
  - 3.1. Using Python as a Calculator
    - 3.1.1. Numbers
    - 3.1.2. Strings
    - 3.1.3. Lists
  - 3.2. First Steps Towards Programming
- **4. More Control Flow Tools**
  - 4.1. *if* Statements
  - 4.2. *for* Statements
  - 4.3. The *range()* Function
  - 4.4. *break* and *continue* Statements, and *else* Clauses on Loops
  - 4.5. *pass* Statements
  - 4.6. Defining Functions
  - 4.7. More on Defining Functions
    - 4.7.1. Default Argument Values
4.7.2. Keyword Arguments
4.7.3. Arbitrary Argument Lists
4.7.4. Unpacking Argument Lists
4.7.5. Lambda Expressions
4.7.6. Documentation Strings
4.7.7. Function Annotations
4.8. Intermezzo: Coding Style

5. Data Structures
5.1. More on Lists
5.1.1. Using Lists as Stacks
5.1.2. Using Lists as Queues
5.1.3. List Comprehensions
5.1.4. Nested List Comprehensions
5.2. The `del` statement
5.3. Tuples and Sequences
5.4. Sets
5.5. Dictionaries
5.6. Looping Techniques
5.7. More on Conditions
5.8. Comparing Sequences and Other Types

6. Modules
6.1. More on Modules
6.1.1. Executing modules as scripts
6.1.2. The Module Search Path
6.1.3. “Compiled” Python files
6.2. Standard Modules
6.3. The `dir()` Function
6.4. Packages
6.4.1. Importing * From a Package
6.4.2. Intra-package References
6.4.3. Packages in Multiple Directories

7. Input and Output
7.1. Fancier Output Formatting
7.1.1. Formatted String Literals
7.1.2. The String format() Method
7.1.3. Manual String Formatting
7.1.4. Old string formatting
7.2. Reading and Writing Files
   7.2.1. Methods of File Objects
   7.2.2. Saving structured data with json

8. Errors and Exceptions
   8.1. Syntax Errors
   8.2. Exceptions
   8.3. Handling Exceptions
   8.4. Raising Exceptions
   8.5. User-defined Exceptions
   8.6. Defining Clean-up Actions
   8.7. Predefined Clean-up Actions

9. Classes
   9.1. A Word About Names and Objects
   9.2. Python Scopes and Namespaces
      9.2.1. Scopes and Namespaces Example
   9.3. A First Look at Classes
      9.3.1. Class Definition Syntax
      9.3.2. Class Objects
      9.3.3. Instance Objects
      9.3.4. Method Objects
      9.3.5. Class and Instance Variables
   9.4. Random Remarks
   9.5. Inheritance
      9.5.1. Multiple Inheritance
   9.6. Private Variables
   9.7. Odds and Ends
   9.8. Iterators
   9.9. Generators
   9.10. Generator Expressions

10. Brief Tour of the Standard Library
    10.1. Operating System Interface
- 10.2. File Wildcards
- 10.3. Command Line Arguments
- 10.4. Error Output Redirection and Program Termination
- 10.5. String Pattern Matching
- 10.6. Mathematics
- 10.7. Internet Access
- 10.8. Dates and Times
- 10.9. Data Compression
- 10.10. Performance Measurement
- 10.11. Quality Control
- 10.12. Batteries Included
- 11. Brief Tour of the Standard Library — Part II
  - 11.1. Output Formatting
  - 11.2. Templating
  - 11.3. Working with Binary Data Record Layouts
  - 11.4. Multi-threading
  - 11.5. Logging
  - 11.6. Weak References
  - 11.7. Tools for Working with Lists
  - 11.8. Decimal Floating Point Arithmetic
- 12. Virtual Environments and Packages
  - 12.1. Introduction
  - 12.2. Creating Virtual Environments
  - 12.3. Managing Packages with pip
- 13. What Now?
- 14. Interactive Input Editing and History Substitution
  - 14.1. Tab Completion and History Editing
  - 14.2. Alternatives to the Interactive Interpreter
- 15. Floating Point Arithmetic: Issues and Limitations
  - 15.1. Representation Error
- 16. Appendix
  - 16.1. Interactive Mode
    - 16.1.1. Error Handling
    - 16.1.2. Executable Python Scripts
- 16.1.3. The Interactive Startup File
- 16.1.4. The Customization Modules
Python Setup and Usage

This part of the documentation is devoted to general information on the setup of the Python environment on different platforms, the invocation of the interpreter and things that make working with Python easier.

- **1. Command line and environment**
  - **1.1. Command line**
    - 1.1.1. Interface options
    - 1.1.2. Generic options
    - 1.1.3. Miscellaneous options
    - 1.1.4. Options you shouldn’t use
  - **1.2. Environment variables**
    - 1.2.1. Debug-mode variables

- **2. Using Python on Unix platforms**
  - **2.1. Getting and installing the latest version of Python**
    - 2.1.1. On Linux
    - 2.1.2. On FreeBSD and OpenBSD
    - 2.1.3. On OpenSolaris
  - **2.2. Building Python**
  - **2.3. Python-related paths and files**
  - **2.4. Miscellaneous**
  - **2.5. Editors and IDEs**

- **3. Using Python on Windows**
  - **3.1. The full installer**
    - 3.1.1. Installation steps
    - 3.1.2. Removing the MAX_PATH Limitation
    - 3.1.3. Installing Without UI
    - 3.1.4. Installing Without Downloading
    - 3.1.5. Modifying an install
  - **3.2. The Microsoft Store package**
    - 3.2.1. Known Issues
- 3.3. The nuget.org packages
- 3.4. The embeddable package
  - 3.4.1. Python Application
  - 3.4.2. Embedding Python
- 3.5. Alternative bundles
- 3.6. Configuring Python
  - 3.6.1. Excursus: Setting environment variables
  - 3.6.2. Finding the Python executable
- 3.7. Python Launcher for Windows
  - 3.7.1. Getting started
    - 3.7.1.1. From the command-line
    - 3.7.1.2. Virtual environments
    - 3.7.1.3. From a script
    - 3.7.1.4. From file associations
  - 3.7.2. Shebang Lines
  - 3.7.3. Arguments in shebang lines
  - 3.7.4. Customization
    - 3.7.4.1. Customization via INI files
    - 3.7.4.2. Customizing default Python versions
  - 3.7.5. Diagnostics
- 3.8. Finding modules
- 3.9. Additional modules
  - 3.9.1. PyWin32
  - 3.9.2. cx_Freeze
  - 3.9.3. WConio
- 3.10. Compiling Python on Windows
- 3.11. Other Platforms
- 4. Using Python on a Macintosh
  - 4.1. Getting and Installing MacPython
    - 4.1.1. How to run a Python script
    - 4.1.2. Running scripts with a GUI
    - 4.1.3. Configuration
  - 4.2. The IDE
  - 4.3. Installing Additional Python Packages
- 4.4. GUI Programming on the Mac
- 4.5. Distributing Python Applications on the Mac
- 4.6. Other Resources
The Python Language Reference

This reference manual describes the syntax and “core semantics” of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in The Python Standard Library. For an informal introduction to the language, see The Python Tutorial. For C or C++ programmers, two additional manuals exist: Extending and Embedding the Python Interpreter describes the high-level picture of how to write a Python extension module, and the Python/C API Reference Manual describes the interfaces available to C/C++ programmers in detail.

- 1. Introduction
  - 1.1. Alternate Implementations
  - 1.2. Notation
- 2. Lexical analysis
  - 2.1. Line structure
  - 2.2. Other tokens
  - 2.3. Identifiers and keywords
  - 2.4. Literals
  - 2.5. Operators
  - 2.6. Delimiters
- 3. Data model
  - 3.1. Objects, values and types
  - 3.2. The standard type hierarchy
  - 3.3. Special method names
  - 3.4. Coroutines
- 4. Execution model
  - 4.1. Structure of a program
  - 4.2. Naming and binding
  - 4.3. Exceptions
5. The import system
   - 5.1. importlib
   - 5.2. Packages
   - 5.3. Searching
   - 5.4. Loading
   - 5.5. The Path Based Finder
   - 5.6. Replacing the standard import system
   - 5.7. Special considerations for __main__
   - 5.8. Open issues
   - 5.9. References

6. Expressions
   - 6.1. Arithmetic conversions
   - 6.2. Atoms
   - 6.3. Primaries
   - 6.4. Await expression
   - 6.5. The power operator
   - 6.6. Unary arithmetic and bitwise operations
   - 6.7. Binary arithmetic operations
   - 6.8. Shifting operations
   - 6.9. Binary bitwise operations
   - 6.10. Comparisons
   - 6.11. Boolean operations
   - 6.12. Conditional expressions
   - 6.13. Lambdas
   - 6.14. Expression lists
   - 6.15. Evaluation order
   - 6.16. Operator precedence

7. Simple statements
   - 7.1. Expression statements
   - 7.2. Assignment statements
   - 7.3. The assert statement
   - 7.4. The pass statement
   - 7.5. The del statement
   - 7.6. The return statement
- 7.7. The `yield` statement
- 7.8. The `raise` statement
- 7.9. The `break` statement
- 7.10. The `continue` statement
- 7.11. The `import` statement
- 7.12. The `global` statement
- 7.13. The `nonlocal` statement

- 8. Compound statements
  - 8.1. The `if` statement
  - 8.2. The `while` statement
  - 8.3. The `for` statement
  - 8.4. The `try` statement
  - 8.5. The `with` statement
  - 8.6. Function definitions
  - 8.7. Class definitions
  - 8.8. Coroutines

- 9. Top-level components
  - 9.1. Complete Python programs
  - 9.2. File input
  - 9.3. Interactive input
  - 9.4. Expression input

- 10. Full Grammar specification
The Python Standard Library

While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

• Introduction
Notes on availability

• **Built-in Functions**
  • **Built-in Constants**
  
  ○ Constants added by the site module

• **Built-in Types**
  • Truth Value Testing
  • Boolean Operations — and, or, not
  • Comparisons
  • Numeric Types — int, float, complex
  • Iterator Types
  • Sequence Types — list, tuple, range
  • Text Sequence Type — str
  • Binary Sequence Types — bytes, bytearray, memoryview
  • Set Types — set, frozenset
  • Mapping Types — dict
  • Context Manager Types
  • Other Built-in Types
  • Special Attributes

• **Built-in Exceptions**
  • Base classes
  • Concrete exceptions
  • Warnings
  • Exception hierarchy

• **Text Processing Services**
  • string — Common string operations
  • re — Regular expression operations
  • difflib — Helpers for computing deltas
  • textwrap — Text wrapping and filling
  • unicodedata — Unicode Database
  • stringprep — Internet String Preparation
  • readline — GNU readline interface
  • rlcompleter — Completion function for GNU readline
• Binary Data Services
  ○ struct — Interpret bytes as packed binary data
  ○ codecs — Codec registry and base classes

• Data Types
  ○ datetime — Basic date and time types
  ○ calendar — General calendar-related functions
  ○ collections — Container datatypes
  ○ collections.abc — Abstract Base Classes for Containers
  ○ heapq — Heap queue algorithm
  ○ bisect — Array bisection algorithm
  ○ array — Efficient arrays of numeric values
  ○ weakref — Weak references
  ○ types — Dynamic type creation and names for built-in types
  ○ copy — Shallow and deep copy operations
  ○ pprint — Data pretty printer
  ○ reprlib — Alternate repr() implementation
  ○ enum — Support for enumerations

• Numeric and Mathematical Modules
  ○ numbers — Numeric abstract base classes
  ○ math — Mathematical functions
  ○ cmath — Mathematical functions for complex numbers
  ○ decimal — Decimal fixed point and floating point arithmetic
  ○ fractions — Rational numbers
  ○ random — Generate pseudo-random numbers
  ○ statistics — Mathematical statistics functions

• Functional Programming Modules
  ○ itertools — Functions creating iterators for efficient looping
  ○ functools — Higher-order functions and operations on callable objects
  ○ operator — Standard operators as functions

• File and Directory Access
- `pathlib` — Object-oriented filesystem paths
- `os.path` — Common pathname manipulations
- `fileinput` — Iterate over lines from multiple input streams
- `stat` — Interpreting `stat()` results
- `filecmp` — File and Directory Comparisons
- `tempfile` — Generate temporary files and directories
- `glob` — Unix style pathname pattern expansion
- `fnmatch` — Unix filename pattern matching
- `linecache` — Random access to text lines
- `shutil` — High-level file operations
- `macpath` — Mac OS 9 path manipulation functions

**Data Persistence**
- `pickle` — Python object serialization
- `copyreg` — Register pickle support functions
- `shelve` — Python object persistence
- `marshal` — Internal Python object serialization
- `dbm` — Interfaces to Unix “databases”
- `sqlite3` — DB-API 2.0 interface for SQLite databases

**Data Compression and Archiving**
- `zlib` — Compression compatible with `gzip`
- `gzip` — Support for `gzip` files
- `bz2` — Support for `bzip2` compression
- `lzma` — Compression using the LZMA algorithm
- `zipfile` — Work with ZIP archives
- `tarfile` — Read and write tar archive files

**File Formats**
- `csv` — CSV File Reading and Writing
- `configparser` — Configuration file parser
- `netrc` — netrc file processing
- `xdrlib` — Encode and decode XDR data
- `plistlib` — Generate and parse Mac OS X `.plist` files

**Cryptographic Services**
• hashlib — Secure hashes and message digests
• hmac — Keyed-Hashing for Message Authentication
• secrets — Generate secure random numbers for managing secrets

• Generic Operating System Services
  • os — Miscellaneous operating system interfaces
  • io — Core tools for working with streams
  • time — Time access and conversions
  • argparse — Parser for command-line options, arguments and sub-commands
  • getopt — C-style parser for command line options
  • logging — Logging facility for Python
  • logging.config — Logging configuration
  • logging.handlers — Logging handlers
  • getpass — Portable password input
  • curses — Terminal handling for character-cell displays
  • curses.textpad — Text input widget for curses programs
  • curses.ascii — Utilities for ASCII characters
  • curses.panel — A panel stack extension for curses
  • platform — Access to underlying platform’s identifying data
  • errno — Standard errno system symbols
  • ctypes — A foreign function library for Python

• Concurrent Execution
  • threading — Thread-based parallelism
  • multiprocessing — Process-based parallelism
  • The concurrent package
  • concurrent.futures — Launching parallel tasks
  • subprocess — Subprocess management
  • sched — Event scheduler
  • queue — A synchronized queue class
  • _thread — Low-level threading API
  • _dummy_thread — Drop-in replacement for the _thread
module
don_{ dummy\_threading} — Drop-in replacement for the threading module

- contextvars — Context Variables
  - Context Variables
  - Manual Context Management
  - asyncio support

- Networking and Interprocess Communication
  - asyncio — Asynchronous I/O
  - socket — Low-level networking interface
  - ssl — TLS/SSL wrapper for socket objects
  - select — Waiting for I/O completion
  - selectors — High-level I/O multiplexing
  - asyncore — Asynchronous socket handler
  - asynchat — Asynchronous socket command/response handler
  - signal — Set handlers for asynchronous events
  - mmap — Memory-mapped file support

- Internet Data Handling
  - email — An email and MIME handling package
  - json — JSON encoder and decoder
  - mailcap — Mailcap file handling
  - mailbox — Manipulate mailboxes in various formats
  - mimetypes — Map filenames to MIME types
  - base64 — Base16, Base32, Base64, Base85 Data Encodings
  - binhex — Encode and decode binhex4 files
  - binascii — Convert between binary and ASCII
  - quopri — Encode and decode MIME quoted-printable data
  - uu — Encode and decode uuencode files

- Structured Markup Processing Tools
  - html — HyperText Markup Language support
- html.parser — Simple HTML and XHTML parser
- html.entities — Definitions of HTML general entities
- XML Processing Modules
- xml.etree.ElementTree — The ElementTree XML API
- xml.dom — The Document Object Model API
- xml.dom.minidom — Minimal DOM implementation
- xml.dom.pulldom — Support for building partial DOM trees
- xml.sax — Support for SAX2 parsers
- xml.sax.handler — Base classes for SAX handlers
- xml.sax.saxutils — SAX Utilities
- xml.sax.xmlreader — Interface for XML parsers
- xml.parsers.expat — Fast XML parsing using Expat

- Internet Protocols and Support
- webbrowser — Convenient Web-browser controller
- cgi — Common Gateway Interface support
- cgitb — Traceback manager for CGI scripts
- wsgiref — WSGI Utilities and Reference Implementation
- urllib — URL handling modules
- urllib.request — Extensible library for opening URLs
- urllib.response — Response classes used by urllib
- urllib.parse — Parse URLs into components
- urllib.error — Exception classes raised by urllib.request
- urllib.robotparser — Parser for robots.txt
- http — HTTP modules
- http.client — HTTP protocol client
- ftplib — FTP protocol client
- poplib — POP3 protocol client
- imaplib — IMAP4 protocol client
- nntplib — NNTP protocol client
- smtplib — SMTP protocol client
- smtpd — SMTP Server
- telnetlib — Telnet client
- uuid — UUID objects according to RFC 4122
- socketserver — A framework for network servers
- http.server — HTTP servers
- http.cookies — HTTP state management
- http.cookiejar — Cookie handling for HTTP clients
- xmlrpc — XMLRPC server and client modules
- xmlrpc.client — XML-RPC client access
- xmlrpc.server — Basic XML-RPC servers
- ipaddress — IPv4/IPv6 manipulation library

- Multimedia Services
  - audioop — Manipulate raw audio data
  - aifc — Read and write AIFF and AIFC files
  - sunau — Read and write Sun AU files
  - wave — Read and write WAV files
  - chunk — Read IFF chunked data
  - colorsys — Conversions between color systems
  - imghdr — Determine the type of an image
  - sndhdr — Determine type of sound file
  - ossaudiodev — Access to OSS-compatible audio devices

- Internationalization
  - gettext — Multilingual internationalization services
  - locale — Internationalization services

- Program Frameworks
  - turtle — Turtle graphics
  - cmd — Support for line-oriented command interpreters
  - shlex — Simple lexical analysis

- Graphical User Interfaces with Tk
  - tkinter — Python interface to Tcl/Tk
  - tkinter.ttk — Tk themed widgets
  - tkinter.tix — Extension widgets for Tk
  - tkinter.scrolledtext — Scrolled Text Widget
- IDLE
- Other Graphical User Interface Packages

### Development Tools
- typing — Support for type hints
- pydoc — Documentation generator and online help system
- doctest — Test interactive Python examples
- unittest — Unit testing framework
- unittest.mock — mock object library
- unittest.mock — getting started
- 2to3 - Automated Python 2 to 3 code translation
- test — Regression tests package for Python
- test.support — Utilities for the Python test suite
- test.support.script_helper — Utilities for the Python execution tests

### Debugging and Profiling
- bdb — Debugger framework
- faulthandler — Dump the Python traceback
- pdb — The Python Debugger
- The Python Profilers
- timeit — Measure execution time of small code snippets
- trace — Trace or track Python statement execution
- tracemalloc — Trace memory allocations

### Software Packaging and Distribution
- distutils — Building and installing Python modules
- ensurepip — Bootstrapping the pip installer
- venv — Creation of virtual environments
- zipapp — Manage executable Python zip archives

### Python Runtime Services
- sys — System-specific parameters and functions
- sysconfig — Provide access to Python’s configuration information
- builtins — Built-in objects
- __main__ — Top-level script environment
- warnings — Warning control
- dataclasses — Data Classes
- contextlib — Utilities for with-statement contexts
- abc — Abstract Base Classes
- atexit — Exit handlers
- traceback — Print or retrieve a stack traceback
- __future__ — Future statement definitions
- gc — Garbage Collector interface
- inspect — Inspect live objects
- site — Site-specific configuration hook

- Custom Python Interpreters
  - code — Interpreter base classes
  - codeop — Compile Python code

- Importing Modules
  - zipimport — Import modules from Zip archives
  - pkgutil — Package extension utility
  - modulefinder — Find modules used by a script
  - runpy — Locating and executing Python modules
  - importlib — The implementation of import

- Python Language Services
  - parser — Access Python parse trees
  - ast — Abstract Syntax Trees
  - symtable — Access to the compiler’s symbol tables
  - symbol — Constants used with Python parse trees
  - token — Constants used with Python parse trees
  - keyword — Testing for Python keywords
  - tokenize — Tokenizer for Python source
  - tabnanny — Detection of ambiguous indentation
  - pyc1br — Python class browser support
  - py_compile — Compile Python source files
  - compileall — Byte-compile Python libraries
- dis — Disassembler for Python bytecode
- pickletools — Tools for pickle developers

- Miscellaneous Services
  - formatter — Generic output formatting

- MS Windows Specific Services
  - msilib — Read and write Microsoft Installer files
  - msvcr — Useful routines from the MS VC++ runtime
  - winreg — Windows registry access
  - winsound — Sound-playing interface for Windows

- Unix Specific Services
  - posix — The most common POSIX system calls
  - pwd — The password database
  - spwd — The shadow password database
  - grp — The group database
  - crypt — Function to check Unix passwords
  - termios — POSIX style tty control
  - tty — Terminal control functions
  - pty — Pseudo-terminal utilities
  - fcntl — The fcntl and ioctl system calls
  - pipes — Interface to shell pipelines
  - resource — Resource usage information
  - nis — Interface to Sun’s NIS (Yellow Pages)
  - syslog — Unix syslog library routines

- Superseded Modules
  - optparse — Parser for command line options
  - imp — Access the import internals

- Undocumented Modules
  - Platform specific modules
Extending and Embedding the Python Interpreter

This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see The Python Tutorial. The Python Language Reference gives a more formal definition of the language. The Python Standard Library documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate Python/C API Reference Manual.
Recommended third party tools

This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools like Cython, cffi, SWIG and Numba offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

See also:

Python Packaging User Guide: Binary Extensions

The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.
Creating extensions without third party tools

This section of the guide covers creating C and C++ extensions without assistance from third party tools. It is intended primarily for creators of those tools, rather than being a recommended way to create your own C extensions.

- **1. Extending Python with C or C++**
  - 1.1. A Simple Example
  - 1.2. Intermezzo: Errors and Exceptions
  - 1.3. Back to the Example
  - 1.4. The Module’s Method Table and Initialization Function
  - 1.5. Compilation and Linkage
  - 1.6. Calling Python Functions from C
  - 1.7. Extracting Parameters in Extension Functions
  - 1.8. Keyword Parameters for Extension Functions
  - 1.9. Building Arbitrary Values
  - 1.10. Reference Counts
  - 1.11. Writing Extensions in C++
  - 1.12. Providing a C API for an Extension Module

- **2. Defining Extension Types: Tutorial**
  - 2.1. The Basics
  - 2.2. Adding data and methods to the Basic example
  - 2.3. Providing finer control over data attributes
  - 2.4. Supporting cyclic garbage collection
  - 2.5. Subclassing other types

- **3. Defining Extension Types: Assorted Topics**
  - 3.1. Finalization and De-allocation
  - 3.2. Object Presentation
  - 3.3. Attribute Management
  - 3.4. Object Comparison
  - 3.5. Abstract Protocol Support
3.6. Weak Reference Support
3.7. More Suggestions

4. Building C and C++ Extensions
   4.1. Building C and C++ Extensions with distutils
   4.2. Distributing your extension modules

5. Building C and C++ Extensions on Windows
   5.1. A Cookbook Approach
   5.2. Differences Between Unix and Windows
   5.3. Using DLLs in Practice
Embedding the CPython runtime in a larger application

Sometimes, rather than creating an extension that runs inside the Python interpreter as the main application, it is desirable to instead embed the CPython runtime inside a larger application. This section covers some of the details involved in doing that successfully.

- **1. Embedding Python in Another Application**
  - **1.1. Very High Level Embedding**
  - **1.2. Beyond Very High Level Embedding: An overview**
  - **1.3. Pure Embedding**
  - **1.4. Extending Embedded Python**
  - **1.5. Embedding Python in C++**
  - **1.6. Compiling and Linking under Unix-like systems**
This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to Extending and Embedding the Python Interpreter, which describes the general principles of extension writing but does not document the API functions in detail.

- Introduction
  - Coding standards
  - Include Files
  - Useful macros
  - Objects, Types and Reference Counts
  - Exceptions
  - Embedding Python
  - Debugging Builds
- Stable Application Binary Interface
- The Very High Level Layer
- Reference Counting
- Exception Handling
  - Printing and clearing
  - Raising exceptions
  - Issuing warnings
  - Querying the error indicator
  - Signal Handling
  - Exception Classes
  - Exception Objects
  - Unicode Exception Objects
  - Recursion Control
  - Standard Exceptions
  - Standard Warning Categories
- Utilities
- Operating System Utilities
- System Functions
- Process Control
- Importing Modules
- Data marshalling support
- Parsing arguments and building values
- String conversion and formatting
- Reflection
- Codec registry and support functions

- Abstract Objects Layer
  - Object Protocol
  - Number Protocol
  - Sequence Protocol
  - Mapping Protocol
  - Iterator Protocol
  - Buffer Protocol
  - Old Buffer Protocol

- Concrete Objects Layer
  - Fundamental Objects
  - Numeric Objects
  - Sequence Objects
  - Container Objects
  - Function Objects
  - Other Objects

- Initialization, Finalization, and Threads
  - Before Python Initialization
  - Global configuration variables
  - Initializing and finalizing the interpreter
  - Process-wide parameters
  - Thread State and the Global Interpreter Lock
  - Sub-interpreter support
  - Asynchronous Notifications
  - Profiling and Tracing
  - Advanced Debugger Support
• Thread Local Storage Support
• **Memory Management**
  • Overview
  • Raw Memory Interface
  • Memory Interface
  • Object allocators
  • Default Memory Allocators
  • Customize Memory Allocators
  • The pymalloc allocator
  • tracemalloc C API
  • Examples
• **Object Implementation Support**
  • Allocating Objects on the Heap
  • Common Object Structures
  • Type Objects
  • Number Object Structures
  • Mapping Object Structures
  • Sequence Object Structures
  • Buffer Object Structures
  • Async Object Structures
  • Supporting Cyclic Garbage Collection
• **API and ABI Versioning**
Distributing Python Modules

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the distribution part of the process. For a guide to installing other Python projects, refer to the installation guide.

**Note:** For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.
Key terms

- the **Python Packaging Index** is a public repository of open source licensed packages made available for use by other Python users.
- the **Python Packaging Authority** are the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation and issue trackers on both GitHub and BitBucket.
- **distutils** is the original build and distribution system first added to the Python standard library in 1998. While direct use of distutils is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).
- **setuptools** is a (largely) drop-in replacement for distutils first published in 2004. Its most notable addition over the unmodified distutils tools was the ability to declare dependencies on other packages. It is currently recommended as a more regularly updated alternative to distutils that offers consistent support for more recent packaging standards across a wide range of Python versions.
- **wheel** (in this context) is a project that adds the bdist_wheel command to distutils/setuptools. This produces a cross platform binary packaging format (called “wheels” or “wheel files” and defined in PEP 427) that allows Python libraries, even those including binary extensions, to be installed on a system without needing to be built locally.
Open source licensing and collaboration

In most parts of the world, software is automatically covered by copyright. This means that other developers require explicit permission to copy, use, modify and redistribute the software.

Open source licensing is a way of explicitly granting such permission in a relatively consistent way, allowing developers to share and collaborate efficiently by making common solutions to various problems freely available. This leaves many developers free to spend more time focusing on the problems that are relatively unique to their specific situation.

The distribution tools provided with Python are designed to make it reasonably straightforward for developers to make their own contributions back to that common pool of software if they choose to do so.

The same distribution tools can also be used to distribute software within an organisation, regardless of whether that software is published as open source software or not.
Installing the tools

The standard library does not include build tools that support modern Python packaging standards, as the core development team has found that it is important to have standard tools that work consistently, even on older versions of Python.

The currently recommended build and distribution tools can be installed by invoking the `pip` module at the command line:

```
python -m pip install setuptools wheel twine
```

**Note:** For POSIX users (including Mac OS X and Linux users), these instructions assume the use of a virtual environment.

For Windows users, these instructions assume that the option to adjust the system PATH environment variable was selected when installing Python.

The Python Packaging User Guide includes more details on the currently recommended tools.
Reading the guide

The Python Packaging User Guide covers the various key steps and elements involved in creating a project:

- Project structure
- Building and packaging the project
- Uploading the project to the Python Packaging Index
How do I…?

These are quick answers or links for some common tasks.

… choose a name for my project?

This isn’t an easy topic, but here are a few tips:

- check the Python Packaging Index to see if the name is already in use
- check popular hosting sites like GitHub, BitBucket, etc to see if there is already a project with that name
- check what comes up in a web search for the name you’re considering
- avoid particularly common words, especially ones with multiple meanings, as they can make it difficult for users to find your software when searching for it

… create and distribute binary extensions?

This is actually quite a complex topic, with a variety of alternatives available depending on exactly what you’re aiming to achieve. See the Python Packaging User Guide for more information and recommendations.

See also: Python Packaging User Guide: Binary Extensions
Installing Python Modules

Email: distutils-sig@python.org

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the installation part of the process. For a guide to creating and sharing your own Python projects, refer to the distribution guide.

Note: For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.
Key terms

- **pip** is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.
- A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.
- **venv** is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing pip into all created virtual environments.
- **virtualenv** is a third party alternative (and predecessor) to venv. It allows virtual environments to be used on versions of Python prior to 3.4, which either don’t provide venv at all, or aren’t able to automatically install pip into created environments.
- The **Python Packaging Index** is a public repository of open source licensed packages made available for use by other Python users.
- the **Python Packaging Authority** is the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation, and issue trackers on both GitHub and BitBucket.
- **distutils** is the original build and distribution system first added to the Python standard library in 1998. While direct use of distutils is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).

*Deprecated since version 3.6:* **pyvenv** was the recommended tool for
creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

*Changed in version 3.5:* The use of `venv` is now recommended for creating virtual environments.

**See also:** Python Packaging User Guide: Creating and using virtual environments
Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

**Note:** For POSIX users (including Mac OS X and Linux users), the examples in this guide assume the use of a virtual environment.

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It's also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as `>`, `<` or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
python -m pip install SomePackage==1.0.4  # specific version
python -m pip install "SomePackage>=1.0.4"  # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade SomePackage
```
More information and resources regarding `pip` and its capabilities can be found in the Python Packaging User Guide.

Creation of virtual environments is done through the `venv` module. Installing packages into an active virtual environment uses the commands shown above.

**See also:** Python Packaging User Guide: Installing Python Distribution Packages
How do I …?

These are quick answers or links for some common tasks.

... install pip in versions of Python prior to Python 3.4?

Python only started bundling pip with Python 3.4. For earlier versions, pip needs to be “bootstrapped” as described in the Python Packaging User Guide.

See also: Python Packaging User Guide: Requirements for Installing Packages

... install packages just for the current user?

Passing the --user option to python -m pip install will install a package just for the current user, rather than for all users of the system.

... install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren’t currently easy to install using pip directly. At this point in time, it will often be easier for users to install these packages by other means rather than attempting to install them with pip.

See also: Python Packaging User Guide: Installing Scientific Packages
... work with multiple versions of Python installed in parallel?

On Linux, Mac OS X, and other POSIX systems, use the versioned Python commands in combination with the `-m` switch to run the appropriate copy of `pip`:

```
python2   -m pip install SomePackage  # default Python 2
python2.7 -m pip install SomePackage  # specifically Python 2.7
python3   -m pip install SomePackage  # default Python 3
python3.4 -m pip install SomePackage  # specifically Python 3.4
```

 Appropriately versioned `pip` commands may also be available.

On Windows, use the `py` Python launcher in combination with the `-m` switch:

```
py -2   -m pip install SomePackage  # default Python 2
py -2.7 -m pip install SomePackage  # specifically Python 2.7
py -3   -m pip install SomePackage  # default Python 3
py -3.4 -m pip install SomePackage  # specifically Python 3.4
```
Common installation issues

Installing into the system Python on Linux

On Linux systems, a Python installation will typically be included as part of the distribution. Installing into this Python installation requires root access to the system, and may interfere with the operation of the system package manager and other components of the system if a component is unexpectedly upgraded using pip.

On such systems, it is often better to use a virtual environment or a per-user installation when installing packages with pip.

Pip not installed

It is possible that pip does not get installed by default. One potential fix is:

```
python -m ensurepip --default-pip
```

There are also additional resources for installing pip.

Installing binary extensions

Python has typically relied heavily on source based distribution, with end users being expected to compile extension modules from source as part of the installation process.

With the introduction of support for the binary wheel format, and the ability to publish wheels for at least Windows and Mac OS X through the Python Packaging Index, this problem is expected to diminish over
time, as users are more regularly able to install pre-built extensions rather than needing to build them themselves.

Some of the solutions for installing scientific software that are not yet available as pre-built wheel files may also help with obtaining other binary extensions without needing to build them locally.

See also: Python Packaging User Guide: Binary Extensions
Python HOWTOs

Python HOWTOs are documents that cover a single, specific topic, and attempt to cover it fairly completely. Modelled on the Linux Documentation Project’s HOWTO collection, this collection is an effort to foster documentation that’s more detailed than the Python Library Reference.

Currently, the HOWTOs are:

- Porting Python 2 Code to Python 3
- Porting Extension Modules to Python 3
- Curses Programming with Python
- Descriptor HowTo Guide
- Functional Programming HOWTO
- Logging HOWTO
- Logging Cookbook
- Regular Expression HOWTO
- Socket Programming HOWTO
- Sorting HOW TO
- Unicode HOWTO
- HOWTO Fetch Internet Resources Using The urllib Package
- Argparse Tutorial
- An introduction to the ipaddress module
- Argument Clinic How-To
- Instrumenting CPython with DTrace and SystemTap
Python Frequently Asked Questions

- General Python FAQ
- Programming FAQ
- Design and History FAQ
- Library and Extension FAQ
- Extending/Embedding FAQ
- Python on Windows FAQ
- Graphic User Interface FAQ
- “Why is Python Installed on my Computer?” FAQ
10. API Reference
10.1. **distutils.core** — Core Distutils functionality

The `distutils.core` module is the only module that needs to be installed to use the Distutils. It provides the `setup()` (which is called from the setup script). Indirectly provides the `distutils.dist.Distribution` and `distutils.cmd.Command` class.

`distutils.core.setup(arguments)`

The basic do-everything function that does most everything you could ever ask for from a Distutils method.

The setup function takes a large number of arguments. These are laid out in the following table.

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the package</td>
<td>a string</td>
</tr>
<tr>
<td>version</td>
<td>The version number of the package; see <code>distutils.version</code></td>
<td>a string</td>
</tr>
<tr>
<td>description</td>
<td>A single line describing the package</td>
<td>a string</td>
</tr>
<tr>
<td>long_description</td>
<td>Longer description of the package</td>
<td>a string</td>
</tr>
<tr>
<td>author</td>
<td>The name of the package author</td>
<td>a string</td>
</tr>
<tr>
<td>author_email</td>
<td>The email address of the package author</td>
<td>a string</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>maintainer</td>
<td>The name of the current maintainer, if different from the author. Note that if the maintainer is provided, distutils will use it as the author in PKG-INFO.</td>
<td>a string</td>
</tr>
<tr>
<td>maintainer_email</td>
<td>The email address of the current maintainer, if different from the author.</td>
<td>a string</td>
</tr>
<tr>
<td>url</td>
<td>A URL for the package (homepage)</td>
<td>a string</td>
</tr>
<tr>
<td>download_url</td>
<td>A URL to download the package</td>
<td>a string</td>
</tr>
<tr>
<td>packages</td>
<td>A list of Python packages that distutils will manipulate</td>
<td>a list of strings</td>
</tr>
<tr>
<td>py_modules</td>
<td>A list of Python modules that distutils will manipulate</td>
<td>a list of strings</td>
</tr>
<tr>
<td>scripts</td>
<td>A list of standalone script files to be built and installed</td>
<td>a list of strings</td>
</tr>
<tr>
<td>ext_modules</td>
<td>A list of Python extensions to be built</td>
<td>a list of instances of <code>distutils.core.Extension</code></td>
</tr>
<tr>
<td>classifiers</td>
<td>A list of categories for the package</td>
<td>a list of strings; valid listed on PyPI.</td>
</tr>
<tr>
<td>distclass</td>
<td>the <code>Distribution</code> class to use</td>
<td>a subclass of <code>distutils.core.Distribution</code></td>
</tr>
<tr>
<td><strong>script_name</strong></td>
<td>The name of the setup.py script - defaults to <code>sys.argv[0]</code></td>
<td>a string</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>script_args</strong></td>
<td>Arguments to supply to the setup script</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>options</strong></td>
<td>default options for the setup script</td>
<td>a dictionary</td>
</tr>
<tr>
<td><strong>license</strong></td>
<td>The license for the package</td>
<td>a string</td>
</tr>
<tr>
<td><strong>keywords</strong></td>
<td>Descriptive meta-data, see PEP 314</td>
<td>a list of strings or a comma-separated string</td>
</tr>
<tr>
<td><strong>platforms</strong></td>
<td></td>
<td>a list of strings or a comma-separated string</td>
</tr>
<tr>
<td><strong>cmdclass</strong></td>
<td>A mapping of command names to <code>Command</code> subclasses</td>
<td>a dictionary</td>
</tr>
<tr>
<td><strong>data_files</strong></td>
<td>A list of data files to install</td>
<td>a list</td>
</tr>
<tr>
<td><strong>package_dir</strong></td>
<td>A mapping of package to directory names</td>
<td>a dictionary</td>
</tr>
</tbody>
</table>

```
import distutils.core

run_setup(script_name[, script_args=None, stop_after='run'])
```

Run a setup script in a somewhat controlled environment, and return the `distutils.dist.Distribution` instance that drives things. This is useful if you need to find out the distribution meta-data (passed as keyword args from `script` to `setup()`), or the contents of the config files or command-line.

`script_name` is a file that will be read and run with `exec()`.
sys.argv[0] will be replaced with script for the duration of the call. script_args is a list of strings; if supplied, sys.argv[1:] will be replaced by script_args for the duration of the call.

stop_after tells setup() when to stop processing; possible values:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Stop after the Distribution instance has been created and populated with the keyword arguments to setup()</td>
</tr>
<tr>
<td>config</td>
<td>Stop after config files have been parsed (and their data stored in the Distribution instance)</td>
</tr>
<tr>
<td>cmdline</td>
<td>Stop after the command-line (sys.argv[1:] or script_args) have been parsed (and the data stored in the Distribution instance.)</td>
</tr>
<tr>
<td>run</td>
<td>Stop after all commands have been run (the same as if setup() had been called in the usual way). This is the default value.</td>
</tr>
</tbody>
</table>

In addition, the distutils.core module exposed a number of classes that live elsewhere.

- Extension from distutils.extension
- Command from distutils.cmd
- Distribution from distutils.dist

A short description of each of these follows, but see the relevant module for the full reference.

class distutils.core.Extension

The Extension class describes a single C or C++ extension module
in a setup script. It accepts the following keyword arguments in its constructor:

<table>
<thead>
<tr>
<th>argument name</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>the full name of the extension, including any packages — ie. <em>not</em> a filename or pathname, but Python dotted name</td>
<td>a string</td>
</tr>
<tr>
<td><strong>sources</strong></td>
<td>list of source filenames, relative to the distribution root (where the setup script lives), in Unix form (slash-separated) for portability. Source files may be C, C++, SWIG (.i), platform-specific resource files, or whatever else is recognized by the <code>build_ext</code> command as source for a Python extension.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>include_dirs</strong></td>
<td>list of directories to search for C/C++ header files (in Unix form for portability)</td>
<td>a list of strings</td>
</tr>
<tr>
<td></td>
<td>list of macros to define; each macro is defined using a 2-tuple <code>(name, value)</code>, where <code>value</code> is either</td>
<td></td>
</tr>
<tr>
<td><strong>define_macros</strong></td>
<td>the string to define it to or None to define it without a particular value (equivalent of <code>#define FOO</code> in source or <code>-DFOO</code> on Unix C compiler command line)</td>
<td>a list of tuples</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><strong>undef_macros</strong></td>
<td>list of macros to redefine explicitly</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>library_dirs</strong></td>
<td>list of directories to search for C/C++ libraries at link time</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>libraries</strong></td>
<td>list of library names (not filenames or paths) to link against</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>runtime_library_dirs</strong></td>
<td>list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded)</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>extra_objects</strong></td>
<td>list of extra files to link with (eg. object files not implied by 'sources', static library that must be explicitly specified, binary resource files, etc.)</td>
<td>a list of strings</td>
</tr>
<tr>
<td></td>
<td>any extra platform- and compiler-specific information to use when compiling the source files in</td>
<td></td>
</tr>
<tr>
<td><strong>extra_compile_args</strong></td>
<td>'sources'. For platforms and compilers where a command line makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.</td>
<td>a list of strings</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td><strong>extra_link_args</strong></td>
<td>any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra_compile_args'.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>export_symbols</strong></td>
<td>list of symbols to be exported from a shared extension. Not used on all platforms, and not generally necessary for Python extensions, which typically export exactly one symbol: init + extension_name.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>depends</strong></td>
<td>list of files that the extension depends on.</td>
<td>a list of strings</td>
</tr>
<tr>
<td><strong>extension_language</strong></td>
<td>extension language (i.e. 'c', 'c++',</td>
<td></td>
</tr>
</tbody>
</table>
### language

'objc'). Will be detected from the source extensions if not provided.

<table>
<thead>
<tr>
<th>optional</th>
<th>specifies that a build failure in the extension should not abort the build process, but simply skip the extension.</th>
</tr>
</thead>
</table>

#### class distutils.core.Distribution

A Distribution describes how to build, install and package up a Python software package.

See the setup() function for a list of keyword arguments accepted by the Distribution constructor. setup() creates a Distribution instance.

*Changed in version 3.7:* Distribution now warns if classifiers, keywords and platforms fields are not specified as a list or a string.

#### class distutils.core.Command

A Command class (or rather, an instance of one of its subclasses) implement a single distutils command.
10.2. `distutils.ccompiler` — CCompiler base class

This module provides the abstract base class for the `CCompiler` classes. A `CCompiler` instance can be used for all the compile and link steps needed to build a single project. Methods are provided to set options for the compiler — macro definitions, include directories, link path, libraries and the like.

This module provides the following functions.

`distutils.ccompiler.gen_lib_options(compiler, library_dirs, runtime_library_dirs, libraries)`
Generate linker options for searching library directories and linking with specific libraries. `libraries` and `library_dirs` are, respectively, lists of library names (not filenames!) and search directories. Returns a list of command-line options suitable for use with some compiler (depending on the two format strings passed in).

`distutils.ccompiler.gen_preprocess_options(macros, include_dirs)`
Generate C pre-processor options (`-D`, `-U`, `-I`) as used by at least two types of compilers: the typical Unix compiler and Visual C++. `macros` is the usual thing, a list of 1- or 2-tuples, where `(name,)` means undefine (`-U`) macro `name`, and `(name, value)` means define (`-D`) macro `name` to `value`. `include_dirs` is just a list of directory names to be added to the header file search path (`-I`). Returns a list of command-line options suitable for either Unix compilers or Visual C++.

`distutils.ccompiler.get_default_compiler(osname,`
Determine the default compiler to use for the given platform.

\textit{osname} should be one of the standard Python OS names (i.e. the ones returned by \texttt{os.name}) and \textit{platform} the common value returned by \texttt{sys.platform} for the platform in question.

The default values are \texttt{os.name} and \texttt{sys.platform} in case the parameters are not given.

\texttt{distutils.ccompiler.\texttt{new_compiler}(plat=None, compiler=None, verbose=0, dry_run=0, force=0)}

Factory function to generate an instance of some \texttt{CCompiler} subclass for the supplied platform/compiler combination. \texttt{plat} defaults to \texttt{os.name} (eg. 'posix', 'nt'), and \texttt{compiler} defaults to the default compiler for that platform. Currently only 'posix' and 'nt' are supported, and the default compilers are “traditional Unix interface” (Unix\texttt{CCompiler} class) and Visual C++ (MSVC\texttt{CCompiler} class). Note that it's perfectly possible to ask for a Unix compiler object under Windows, and a Microsoft compiler object under Unix—if you supply a value for \texttt{compiler}, \texttt{plat} is ignored.

\texttt{distutils.ccompiler.\texttt{show_compilers}()}

Print list of available compilers (used by the --help-compiler options to \texttt{build}, \texttt{build_ext}, \texttt{build_clib}).

\texttt{class distutils.ccompiler.\texttt{CCompiler}([verbose=0, dry_run=0, force=0])}

The abstract base class \texttt{CCompiler} defines the interface that must be implemented by real compiler classes. The class also has some utility methods used by several compiler classes.
The basic idea behind a compiler abstraction class is that each instance can be used for all the compile/link steps in building a single project. Thus, attributes common to all of those compile and link steps — include directories, macros to define, libraries to link against, etc. — are attributes of the compiler instance. To allow for variability in how individual files are treated, most of those attributes may be varied on a per-compilation or per-link basis.

The constructor for each subclass creates an instance of the Compiler object. Flags are *verbose* (show verbose output), *dry_run* (don’t actually execute the steps) and *force* (rebuild everything, regardless of dependencies). All of these flags default to 0 (off). Note that you probably don’t want to instantiate `CCompiler` or one of its subclasses directly - use the `distutils.CCompiler.new_compiler()` factory function instead.

The following methods allow you to manually alter compiler options for the instance of the Compiler class.

**add_include_dir(dir)**

Add `dir` to the list of directories that will be searched for header files. The compiler is instructed to search directories in the order in which they are supplied by successive calls to `add_include_dir()`.

**set_include_dirs(dirs)**

Set the list of directories that will be searched to `dirs` (a list of strings). Overrides any preceding calls to `add_include_dir()`; subsequent calls to `add_include_dir()` add to the list passed to `set_include_dirs()`. This does not affect any list of
standard include directories that the compiler may search by default.

**add_library**(libname)
Add *libname* to the list of libraries that will be included in all links driven by this compiler object. Note that *libname* should *not* be the name of a file containing a library, but the name of the library itself: the actual filename will be inferred by the linker, the compiler, or the compiler class (depending on the platform).

The linker will be instructed to link against libraries in the order they were supplied to **add_library()** and/or **set_libraries()**. It is perfectly valid to duplicate library names; the linker will be instructed to link against libraries as many times as they are mentioned.

**set_libraries**(libnames)
Set the list of libraries to be included in all links driven by this compiler object to *libnames* (a list of strings). This does not affect any standard system libraries that the linker may include by default.

**add_library_dir**(dir)
Add *dir* to the list of directories that will be searched for libraries specified to **add_library()** and **set_libraries()**. The linker will be instructed to search for libraries in the order they are supplied to **add_library_dir()** and/or **set_library_dirs()**.

**set_library_dirs**(dirs)
Set the list of library search directories to *dirs* (a list of strings). This does not affect any standard library search path that the
linker may search by default.

**add_runtime_library_dir(dir)**

Add dir to the list of directories that will be searched for shared libraries at runtime.

**set_runtime_library_dirs(dirs)**

Set the list of directories to search for shared libraries at runtime to dirs (a list of strings). This does not affect any standard search path that the runtime linker may search by default.

**define_macro(name[, value=None])**

Define a preprocessor macro for all compilations driven by this compiler object. The optional parameter value should be a string; if it is not supplied, then the macro will be defined without an explicit value and the exact outcome depends on the compiler used.

**undefine_macro(name)**

Undefine a preprocessor macro for all compilations driven by this compiler object. If the same macro is defined by define_macro() and undefined by undefined_macro() the last call takes precedence (including multiple redefinitions or undefinitions). If the macro is redefined/undefined on a per-compilation basis (ie. in the call to compile()), then that takes precedence.

**add_link_object(object)**

Add object to the list of object files (or analogues, such as explicitly named library files or the output of “resource compilers”) to be included in every link driven by this compiler.
functionality similar to GNU autoconf.

detect_language(sources)
Detect the language of a given file, or list of files. Uses the instance attributes language_map (a dictionary), and language_order (a list) to do the job.

find_library_file(dirs, lib[, debug=0])
Search the specified list of directories for a static or shared library file lib and return the full path to that file. If debug is true, look for a debugging version (if that makes sense on the current platform). Return None if lib wasn’t found in any of the specified directories.

has_function(funcname[, includes=include_dirs, l libraries=library_dirs])
Return a boolean indicating whether funcname is supported on the current platform. The optional arguments can be used to augment the compilation environment by providing additional include files and paths and libraries and paths.

library_dir_option(dir)
Return the compiler option to add dir to the list of directories
searched for libraries.

**library_option**(lib)

Return the compiler option to add *lib* to the list of libraries linked into the shared library or executable.

**runtime_library_dir_option**(dir)

Return the compiler option to add *dir* to the list of directories searched for runtime libraries.

**set_executables**(**args**)

Define the executables (and options for them) that will be run to perform the various stages of compilation. The exact set of executables that may be specified here depends on the compiler class (via the ‘executables’ class attribute), but most will have:

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>compiler</em></td>
<td>the C/C++ compiler</td>
</tr>
<tr>
<td><em>linker_so</em></td>
<td>linker used to create shared objects and libraries</td>
</tr>
<tr>
<td><em>linker_exe</em></td>
<td>linker used to create binary executables</td>
</tr>
<tr>
<td><em>archiver</em></td>
<td>static library creator</td>
</tr>
</tbody>
</table>

On platforms with a command-line (Unix, DOS/Windows), each of these is a string that will be split into executable name and (optional) list of arguments. (Splitting the string is done similarly to how Unix shells operate: words are delimited by spaces, but quotes and backslashes can override this. See `distutils.util.split_quoted()`.)

The following methods invoke stages in the build process.
compile(sources[, output_dir=None, macros=None, include_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, depends=None])

Compile one or more source files. Generates object files (e.g. transforms a .c file to a .o file.)

sources must be a list of filenames, most likely C/C++ files, but in reality anything that can be handled by a particular compiler and compiler class (eg. MSVCCompiler can handle resource files in sources). Return a list of object filenames, one per source filename in sources. Depending on the implementation, not all source files will necessarily be compiled, but all corresponding object filenames will be returned.

If output_dir is given, object files will be put under it, while retaining their original path component. That is, foo/bar.c normally compiles to foo/bar.o (for a Unix implementation); if output_dir is build, then it would compile to build/foo/bar.o.

macros, if given, must be a list of macro definitions. A macro definition is either a (name, value) 2-tuple or a (name,) 1-tuple. The former defines a macro; if the value is None, the macro is defined without an explicit value. The 1-tuple case undefines a macro. Later definitions/redefinitions/undefinitions take precedence.

include_dirs, if given, must be a list of strings, the directories to add to the default include file search path for this compilation only.

debug is a boolean; if true, the compiler will be instructed to
output debug symbols in (or alongside) the object file(s).

`extra_preargs` and `extra_postargs` are implementation-dependent. On platforms that have the notion of a command-line (e.g. Unix, DOS/Windows), they are most likely lists of strings: extra command-line arguments to prepend/append to the compiler command line. On other platforms, consult the implementation class documentation. In any event, they are intended as an escape hatch for those occasions when the abstract compiler framework doesn’t cut the mustard.

`depends`, if given, is a list of filenames that all targets depend on. If a source file is older than any file in depends, then the source file will be recompiled. This supports dependency tracking, but only at a coarse granularity.

Raises CompileError on failure.

**create_static_lib**

```python
create_static_lib(objects, output_libname[, output_dir=None, debug=0, target_lang=None])
```

Link a bunch of stuff together to create a static library file. The “bunch of stuff” consists of the list of object files supplied as `objects`, the extra object files supplied to `add_link_object()` and/or `set_link_objects()`, the libraries supplied to `add_library()` and/or `set_libraries()`, and the libraries supplied as `libraries` (if any).

`output_libname` should be a library name, not a filename; the filename will be inferred from the library name. `output_dir` is the directory where the library file will be put.

`debug` is a boolean; if true, debugging information will be included in the library (note that on most platforms, it is the
compile step where this matters: the *debug* flag is included here just for consistency).

*target_lang* is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises LibError on failure.

```python
link(target_desc, objects, output_filename[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])
```

Link a bunch of stuff together to create an executable or shared library file.

The “bunch of stuff” consists of the list of object files supplied as *objects*. *output_filename* should be a filename. If *output_dir* is supplied, *output_filename* is relative to it (i.e. *output_filename* can provide directory components if needed).

*libraries* is a list of libraries to link against. These are library names, not filenames, since they're translated into filenames in a platform-specific way (eg. *foo* becomes *libfoo.a* on Unix and *foo.lib* on DOS/Windows). However, they can include a directory component, which means the linker will look in that specific directory rather than searching all the normal locations.

*library_dirs*, if supplied, should be a list of directories to search for libraries that were specified as bare library names (ie. no directory component). These are on top of the system default and those supplied to *add_library_dir()* and/or *set_library_dirs()*. *runtime_library_dirs* is a list of
directories that will be embedded into the shared library and used to search for other shared libraries that *it* depends on at run-time. (This may only be relevant on Unix.)

`export_symbols` is a list of symbols that the shared library will export. (This appears to be relevant only on Windows.)

`debug` is as for `compile()` and `create_static_lib()`, with the slight distinction that it actually matters on most platforms (as opposed to `create_static_lib()`, which includes a `debug` flag mostly for form's sake).

`extra_preargs` and `extra_postargs` are as for `compile()` (except of course that they supply command-line arguments for the particular linker being used).

`target_lang` is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises `LinkError` on failure.

```python
link_executable(objects, output_progname[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, target_lang=None])
```

Link an executable. `output_progname` is the name of the file executable, while `objects` are a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
link_shared_lib(objects, output_libname[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, ```
Extra parameters are `extra_preargs`, `extra_postargs`, `build_temp`, and `target_lang`.

Link a shared library. `output_libname` is the name of the output library, while `objects` is a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
link_shared_object(objects, output_filename[, output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None])
```

Link a shared object. `output_filename` is the name of the shared object that will be created, while `objects` is a list of object filenames to link in. Other arguments are as for the `link()` method.

```python
preprocess(source[, output_file=None, macros=None, include_dirs=None, extra_preargs=None, extra_postargs=None])
```

Preprocess a single C/C++ source file, named in `source`. Output will be written to file named `output_file`, or `stdout` if `output_file` not supplied. `macros` is a list of macro definitions as for `compile()`, which will augment the macros set with `define_macro()` and `undefine_macro()`. `include_dirs` is a list of directory names that will be added to the default list, in the same way as `add_include_dir()`.

Raises `PreprocessError` on failure.

The following utility methods are defined by the `CCompiler` class, for use by the various concrete subclasses.
**executable_filename** *(basename[, strip_dir=0, output_dir=‘‘]*)

Returns the filename of the executable for the given *basename*. Typically for non-Windows platforms this is the same as the basename, while Windows will get a `.exe` added.

**library_filename** *(libname[, lib_type='static', strip_dir=0, output_dir=‘‘]*)

Returns the filename for the given library name on the current platform. On Unix a library with *lib_type* of 'static' will typically be of the form `liblibname.a`, while a *lib_type* of 'dynamic' will be of the form `liblibname.so`.

**object_filenames** *(source_filenames[, strip_dir=0, output_dir=‘‘]*)

Returns the name of the object files for the given source files. *source_filenames* should be a list of filenames.

**shared_object_filename** *(basename[, strip_dir=0, output_dir=‘‘]*)

Returns the name of a shared object file for the given file name *basename*.

**execute** *(func, args[, msg=None, level=1]*)

Invokes `distutils.util.execute()` . This method invokes a Python function *func* with the given arguments *args*, after logging and taking into account the *dry_run* flag.

**spawn** *(cmd)*

Invokes `distutils.util.spawn()` . This invokes an
external process to run the given command.

**mkpath**(name[, mode=511])

Invokes distutils.dir_util.mkpath(). This creates a directory and any missing ancestor directories.

**move_file**(src, dst)

Invokes distutils.file_util.move_file(). Renames src to dst.

**announce**(msg[, level=1])

Write a message using distutils.log.debug().

**warn**(msg)

Write a warning message msg to standard error.

**debug_print**(msg)

If the debug flag is set on this CCompiler instance, print msg to standard output, otherwise do nothing.
10.3. \texttt{distutils.unixccompiler} — Unix C Compiler

This module provides the UnixCCompiler class, a subclass of CCompiler that handles the typical Unix-style command-line C compiler:

- macros defined with \texttt{-Dname[=value]}
- macros undefined with \texttt{-Uname}
- include search directories specified with \texttt{-I\text{dir}}
- libraries specified with \texttt{-l\text{lib}}
- library search directories specified with \texttt{-L\text{dir}}
- compile handled by \texttt{cc} (or similar) executable with \texttt{-c} option: compiles .c to .o
- link static library handled by \texttt{ar} command (possibly with \texttt{ranlib})
- link shared library handled by \texttt{cc} -shared
10.4. distutils.msvccompiler — Microsoft Compiler

This module provides MSVCCompiler, an implementation of the abstract CCompiler class for Microsoft Visual Studio. Typically, extension modules need to be compiled with the same compiler that was used to compile Python. For Python 2.3 and earlier, the compiler was Visual Studio 6. For Python 2.4 and 2.5, the compiler is Visual Studio .NET 2003.

MSVCCompiler will normally choose the right compiler, linker etc. on its own. To override this choice, the environment variables DISTUTILS_USE_SDK and MSSdk must be both set. MSSdk indicates that the current environment has been setup by the SDK’s SetEnv.Cmd script, or that the environment variables had been registered when the SDK was installed; DISTUTILS_USE_SDK indicates that the distutils user has made an explicit choice to override the compiler selection by MSVCCompiler.
10.5. `distutils.bcppcompiler` — Borland Compiler

This module provides `BorlandCCompiler`, a subclass of the abstract `CCompiler` class for the Borland C++ compiler.
10.6. distutils.cygwincompiler — Cygwin Compiler

This module provides the CygwinCCompiler class, a subclass of UnixCCompiler that handles the Cygwin port of the GNU C compiler to Windows. It also contains the Mingw32CCompiler class which handles the mingw32 port of GCC (same as cygwin in no-cygwin mode).
10.7. `distutils.archive_util` — Archiving utilities

This module provides a few functions for creating archive files, such as tarballs or zipfiles.

`distutils.archive_util.make_archive(base_name, format[, root_dir=None, base_dir=None, verbose=0, dry_run=0])`

Create an archive file (eg. zip or tar). `base_name` is the name of the file to create, minus any format-specific extension; `format` is the archive format: one of zip, tar, gztar, bztar, xztar, or ztar. `root_dir` is a directory that will be the root directory of the archive; ie. we typically `chdir` into `root_dir` before creating the archive. `base_dir` is the directory where we start archiving from; ie. `base_dir` will be the common prefix of all files and directories in the archive. `root_dir` and `base_dir` both default to the current directory. Returns the name of the archive file.

`Changed in version 3.5:` Added support for the xztar format.

`distutils.archive_util.make_tarball(base_name, base_dir[, compress='gzip', verbose=0, dry_run=0])`

Create an (optional compressed) archive as a tar file from all files in and under `base_dir`. `compress` must be 'gzip' (the default), 'bzip2', 'xz', 'compress', or None. For the 'compress' method the compression utility named by `compress` must be on the default program search path, so this is probably Unix-specific. The output tar file will be named `base_dir.tar`, possibly plus the appropriate compression extension (.gz, .bz2, .xz or .Z). Return the output filename.
 Changed in version 3.5: Added support for the xz compression.

distutils.archive_util.make_zipfile(base_name, base_dir[, verbose=0, dry_run=0])

Create a zip file from all files in and under base_dir. The output zip file will be named base_name + .zip. Uses either the zipfile Python module (if available) or the InfoZIP zip utility (if installed and found on the default search path). If neither tool is available, raises DistutilsExecError. Returns the name of the output zip file.
10.8. distutils.dep_util —
Dependency checking

This module provides functions for performing simple, timestamp-based dependency of files and groups of files; also, functions based entirely on such timestamp dependency analysis.

```
distutils.dep_util.newer(source, target)
```

Return true if `source` exists and is more recently modified than `target`, or if `source` exists and `target` doesn't. Return false if both exist and `target` is the same age or newer than `source`. Raise DistutilsFileError if `source` does not exist.

```
distutils.dep_util.newer_pairwise(sources, targets)
```

Walk two filename lists in parallel, testing if each source is newer than its corresponding target. Return a pair of lists (`sources`, `targets`) where source is newer than target, according to the semantics of `newer()`.

```
distutils.dep_util.newer_group(sources, target[, missing='error'])
```

Return true if `target` is out-of-date with respect to any file listed in `sources`. In other words, if `target` exists and is newer than every file in `sources`, return false; otherwise return true. `missing` controls what we do when a source file is missing; the default (`'error'`) is to blow up with an OSError from inside `os.stat()`; if it is `'ignore'`, we silently drop any missing source files; if it is `'newer'`, any missing source files make us assume that `target` is out-of-date (this is handy in “dry-run” mode: it’ll make you pretend to carry out commands that wouldn’t work because inputs are
missing, but that doesn't matter because you're not actually going to run the commands).
10.9. **distutils.dir_util** — Directory tree operations

This module provides functions for operating on directories and trees of directories.

```python
distutils.dir_util.mkpath(name[, mode=0o777, verbose=0, dry_run=0])
```

Create a directory and any missing ancestor directories. If the directory already exists (or if name is the empty string, which means the current directory, which of course exists), then do nothing. Raise DistutilsFileError if unable to create some directory along the way (eg. some sub-path exists, but is a file rather than a directory). If verbose is true, print a one-line summary of each mkdir to stdout. Return the list of directories actually created.

```python
distutils.dir_util.create_tree(base_dir, files[, mode=0o777, verbose=0, dry_run=0])
```

Create all the empty directories under base_dir needed to put files there. base_dir is just the name of a directory which doesn't necessarily exist yet; files is a list of filenames to be interpreted relative to base_dir. base_dir + the directory portion of every file in files will be created if it doesn't already exist. mode, verbose and dry_run flags are as for mkpath().

```python
distutils.dir_util.copy_tree(src, dst[, preserve_mode=1, preserve_times=1, preserve_syalinks=0, update=0, verbose=0, dry_run=0])
```

Copy an entire directory tree src to a new location dst. Both src and
dst must be directory names. If src is not a directory, raise DistutilsFileError. If dst does not exist, it is created with mkpath(). The end result of the copy is that every file in src is copied to dst, and directories under src are recursively copied to dst. Return the list of files that were copied or might have been copied, using their output name. The return value is unaffected by update or dry_run: it is simply the list of all files under src, with the names changed to be under dst.

preserve_mode and preserve_times are the same as for distutils.file_util.copy_file(); note that they only apply to regular files, not to directories. If preserve_symlinks is true, symlinks will be copied as symlinks (on platforms that support them!); otherwise (the default), the destination of the symlink will be copied. update and verbose are the same as for copy_file().

Files in src that begin with .nfs are skipped (more information on these files is available in answer D2 of the NFS FAQ page).

Changed in version 3.3.1: NFS files are ignored.

distutils.dir_util.remove_tree(directory[, verbose=0, dry_run=0])

Recursively remove directory and all files and directories underneath it. Any errors are ignored (apart from being reported to sys.stdout if verbose is true).
10.10. distutils.file_util — Single file operations

This module contains some utility functions for operating on individual files.

\[ \text{distutils.file_util.}\text{copy_file}(\text{src, dst[, preserve_mode=1, preserve_times=1, update=0, link=\text{None, verbose=0, dry_run=0}]}) \]

Copy file \textit{src} to \textit{dst}. If \textit{dst} is a directory, then \textit{src} is copied there with the same name; otherwise, it must be a filename. (If the file exists, it will be ruthlessly clobbered.) If \textit{preserve_mode} is true (the default), the file’s mode (type and permission bits, or whatever is analogous on the current platform) is copied. If \textit{preserve_times} is true (the default), the last-modified and last-access times are copied as well. If \textit{update} is true, \textit{src} will only be copied if \textit{dst} does not exist, or if \textit{dst} does exist but is older than \textit{src}.

\textit{link} allows you to make hard links (using \textit{os.link()}) or symbolic links (using \textit{os.symlink()}) instead of copying: set it to \texttt{'hard'} or \texttt{'sym'}; if it is \texttt{None} (the default), files are copied. Don’t set \textit{link} on systems that don’t support it: \textit{copy_file()} doesn’t check if hard or symbolic linking is available. It uses _copy_file_contents() to copy file contents.

Return a tuple (\textit{dest_name, copied}): \textit{dest_name} is the actual name of the output file, and \textit{copied} is true if the file was copied (or would have been copied, if \textit{dry_run} true).

\[ \text{distutils.file_util.}\text{move_file}(\text{src, dst[, verbose, dry_run]}) \]
Move file src to dst. If dst is a directory, the file will be moved into it with the same name; otherwise, src is just renamed to dst. Returns the new full name of the file.

**Warning:** Handles cross-device moves on Unix using `copy_file()`. What about other systems?

distutils.file_util.**write_file**(filename, contents)

Create a file called filename and write contents (a sequence of strings without line terminators) to it.
This module contains other assorted bits and pieces that don’t fit into any other utility module.

**distutils.util.get_platform()**

Return a string that identifies the current platform. This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by `os.uname()`), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn’t particularly important.

Examples of returned values:

- **linux-i586**
- **linux-alpha**
- **solaris-2.6-sun4u**

For non-POSIX platforms, currently just returns `sys.platform`.

For Mac OS X systems the OS version reflects the minimal version on which binaries will run (that is, the value of `MACOSX_DEPLOYMENT_TARGET` during the build of Python), not the OS version of the current system.

For universal binary builds on Mac OS X the architecture value reflects the universal binary status instead of the architecture of the current processor. For 32-bit universal binaries the architecture is **fat**, for 64-bit universal binaries the architecture is **fat64**, and for 4-way universal binaries the architecture is **universal**. Starting
from Python 2.7 and Python 3.2 the architecture fat3 is used for a 3-way universal build (ppc, i386, x86_64) and intel is used for a universal build with the i386 and x86_64 architectures

Examples of returned values on Mac OS X:

- macOSx-10.3-ppc
- macOSx-10.3-fat
- macOSx-10.5-universal
- macOSx-10.6-intel

\texttt{distutils.util.convert\_path(pathname)}

Return ‘pathname’ as a name that will work on the native filesystem, i.e. split it on ‘/’ and put it back together again using the current directory separator. Needed because filenames in the setup script are always supplied in Unix style, and have to be converted to the local convention before we can actually use them in the filesystem. Raises \texttt{ValueError} on non-Unix-ish systems if \texttt{pathname} either starts or ends with a slash.

\texttt{distutils.util.change\_root(new\_root, pathname)}

Return \texttt{pathname} with \texttt{new\_root} prepended. If \texttt{pathname} is relative, this is equivalent to \texttt{os.path.join(new\_root, pathname)} Otherwise, it requires making \texttt{pathname} relative and then joining the two, which is tricky on DOS/Windows.

\texttt{distutils.util.check\_environ()}

Ensure that ‘os.environ’ has all the environment variables we guarantee that users can use in config files, command-line options, etc. Currently this includes:

- \texttt{HOME} - user's home directory (Unix only)
- PLAT - description of the current platform, including hardware and OS (see `get_platform()`)

distutils.util.subst_vars(s, local_vars)
Perform shell/Perl-style variable substitution on s. Every occurrence of $ followed by a name is considered a variable, and variable is substituted by the value found in the `local_vars` dictionary, or in os.environ if it's not in `local_vars`. os.environ is first checked/augmented to guarantee that it contains certain values: see `check_environ()`. Raise `ValueError` for any variables not found in either `local_vars` or `os.environ`.

Note that this is not a fully-fledged string interpolation function. A valid $variable can consist only of upper and lower case letters, numbers and an underscore. No {{}} or () style quoting is available.

distutils.util.split_quoted(s)
Split a string up according to Unix shell-like rules for quotes and backslashes. In short: words are delimited by spaces, as long as those spaces are not escaped by a backslash, or inside a quoted string. Single and double quotes are equivalent, and the quote characters can be backslash-escaped. The backslash is stripped from any two-character escape sequence, leaving only the escaped character. The quote characters are stripped from any quoted string. Returns a list of words.

distutils.util.execute(func, args=[], msg=None, verbose=0, dry_run=0)
Perform some action that affects the outside world (for instance, writing to the filesystem). Such actions are special because they are disabled by the `dry_run` flag. This method takes care of all that bureaucracy for you; all you have to do is supply the function to call
and an argument tuple for it (to embody the “external action” being performed), and an optional message to print.

```
and a tuple for it (to embody the “external action” being performed), and an optional message to print.

distutils.util.strtobool(val)
```

Convert a string representation of truth to true (1) or false (0).

```
True values are y, yes, t, true, on and 1; false values are n, no, f, false, off and 0. Raises ValueError if val is anything else.
```

```
distutils.util.byte_compile(py_files[, optimize=0, force=0, prefix=None, base_dir=None, verbose=1, dry_run=0, direct=None])
```

Byte-compile a collection of Python source files to .pyc files in a __pycache__ subdirectory (see PEP 3147 and PEP 488). py_files is a list of files to compile; any files that don’t end in .py are silently skipped. optimize must be one of the following:

- 0 - don’t optimize
- 1 - normal optimization (like python -O)
- 2 - extra optimization (like python -OO)

If force is true, all files are recompiled regardless of timestamps.

The source filename encoded in each bytecode file defaults to the filenames listed in py_files; you can modify these with prefix and basedir. prefix is a string that will be stripped off of each source filename, and base_dir is a directory name that will be prepended (after prefix is stripped). You can supply either or both (or neither) of prefix and base_dir, as you wish.

If dry_run is true, doesn’t actually do anything that would affect the filesystem.
Byte-compilation is either done directly in this interpreter process with the standard `py_compile` module, or indirectly by writing a temporary script and executing it. Normally, you should let `byte_compile()` figure out to use direct compilation or not (see the source for details). The `direct` flag is used by the script generated in indirect mode; unless you know what you’re doing, leave it set to `None`.

*Changed in version 3.2.3:* Create `.pyc` files with an `import magic tag` in their name, in a `__pycache__` subdirectory instead of files without tag in the current directory.

*Changed in version 3.5:* Create `.pyc` files according to PEP 488.

`distutils.util.rfc822_escape(header)`

Return a version of `header` escaped for inclusion in an RFC 822 header, by ensuring there are 8 spaces space after each newline. Note that it does no other modification of the string.
10.12. distutils.dist — The Distribution class

This module provides the Distribution class, which represents the module distribution being built/installed/distributed.
This module provides the `Extension` class, used to describe C/C++ extension modules in setup scripts.
10.14. `distutils.debug` — Distutils debug mode

This module provides the DEBUG flag.
10.15. `distutils.errors` — Distutils exceptions

Provides exceptions used by the Distutils modules. Note that Distutils modules may raise standard exceptions; in particular, SystemExit is usually raised for errors that are obviously the end-user’s fault (eg. bad command-line arguments).

This module is safe to use in `from ... import *` mode; it only exports symbols whose names start with `Distutils` and end with `Error`.
10.16. `distutils.fancy_getopt` —
Wrapper around the standard getopt module

This module provides a wrapper around the standard `getopt` module that provides the following additional features:

- short and long options are tied together
- options have help strings, so `fancy_getopt()` could potentially create a complete usage summary
- options set attributes of a passed-in object
- boolean options can have “negative aliases” — eg. if `--quiet` is the “negative alias” of `--verbose`, then `--quiet` on the command line sets `verbose` to false.

```
```

`distutils.fancy_getopt.fancy_getopt(options, negative_opt, object, args)`
Wrapper function. `options` is a list of `(long_option, short_option, help_string)` 3-tuples as described in the constructor for `FancyGetopt`. `negative_opt` should be a dictionary mapping option names to option names, both the key and value should be in the `options` list. `object` is an object which will be used to store values (see the `getopt()` method of the `FancyGetopt` class). `args` is the argument list. Will use `sys.argv[1:]` if you pass `None` as `args`.

```
```

`distutils.fancy_getopt.wrap_text(text, width)`
Wraps `text` to less than `width` wide.

```
```

`class` 

`distutils.fancy_getopt.FancyGetopt([option_table=None])`;
The option_table is a list of 3-tuples: (long_option, short_option, help_string)

If an option takes an argument, its long_option should have '=' appended; short_option should just be a single character, no ':' in any case. short_option should be None if a long_option doesn't have a corresponding short_option. All option tuples must have long options.

The FancyGetopt class provides the following methods:

FancyGetopt.getopt([args=None, object=None])
Parse command-line options in args. Store as attributes on object.

If args is None or not supplied, uses sys.argv[1:]. If object is None or not supplied, creates a new OptionDummy instance, stores option values there, and returns a tuple (args, object). If object is supplied, it is modified in place and getopt() just returns args; in both cases, the returned args is a modified copy of the passed-in args list, which is left untouched.

FancyGetopt.get_option_order()
Returns the list of (option, value) tuples processed by the previous run of getopt() Raises RuntimeError if getopt() hasn't been called yet.

FancyGetopt.generate_help([header=None])
Generate help text (a list of strings, one per suggested line of output) from the option table for this FancyGetopt object.

If supplied, prints the supplied header at the top of the help.
10.17. distutils.filelist — The FileList class

This module provides the FileList class, used for poking about the filesystem and building lists of files.
10.18. **distutils.log** — Simple PEP 282-style logging
10.19. `distutils.spawn` — Spawn a sub-process

This module provides the `spawn()` function, a front-end to various platform-specific functions for launching another program in a sub-process. Also provides `find_executable()` to search the path for a given executable name.
10.20. **distutils.sysconfig** — System configuration information

The **distutils.sysconfig** module provides access to Python's low-level configuration information. The specific configuration variables available depend heavily on the platform and configuration. The specific variables depend on the build process for the specific version of Python being run; the variables are those found in the **Makefile** and configuration header that are installed with Python on Unix systems. The configuration header is called **pyconfig.h** for Python versions starting with 2.2, and **config.h** for earlier versions of Python.

Some additional functions are provided which perform some useful manipulations for other parts of the **distutils** package.

**distutils.sysconfig.** **PREFIX**

The result of `os.path.normpath(sys.prefix)`.

**distutils.sysconfig.** **EXEC_PREFIX**

The result of `os.path.normpath(sys.exec_prefix)`.

**distutils.sysconfig.** **get_config_var**(name)

Return the value of a single variable. This is equivalent to `get_config_vars().get(name)`.

**distutils.sysconfig.** **get_config_vars**(…)

Return a set of variable definitions. If there are no arguments, this returns a dictionary mapping names of configuration variables to values. If arguments are provided, they should be strings, and the return value will be a sequence giving the associated values. If a
given name does not have a corresponding value, `None` will be included for that variable.

distutils.sysconfig.get_config_h_filename()

Return the full path name of the configuration header. For Unix, this will be the header generated by the `configure` script; for other platforms the header will have been supplied directly by the Python source distribution. The file is a platform-specific text file.

distutils.sysconfig.get_makefile_filename()

Return the full path name of the Makefile used to build Python. For Unix, this will be a file generated by the `configure` script; the meaning for other platforms will vary. The file is a platform-specific text file, if it exists. This function is only useful on POSIX platforms.

distutils.sysconfig.get_python_inc([plat_specific[, prefix]])

Return the directory for either the general or platform-dependent C include files. If `plat_specific` is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If `prefix` is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if `plat_specific` is true.

distutils.sysconfig.get_python_lib([plat_specific[, standard_lib[, prefix]]])

Return the directory for either the general or platform-dependent library installation. If `plat_specific` is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If `prefix` is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of
The following function is only intended for use within the `distutils` package.

```
distutils.sysconfig.customize_compiler(compiler)
```

Do any platform-specific customization of a `distutils.ccompiler.CCompiler` instance.

This function is only needed on Unix at this time, but should be called consistently to support forward-compatibility. It inserts the information that varies across Unix flavors and is stored in Python’s Makefile. This information includes the selected compiler, compiler and linker options, and the extension used by the linker for shared objects.

This function is even more special-purpose, and should only be used from Python’s own build procedures.

```
distutils.sysconfig.set_python_build()
```

Inform the `distutils.sysconfig` module that it is being used as part of the build process for Python. This changes a lot of relative locations for files, allowing them to be located in the build area rather than in an installed Python.
10.21. `distutils.text_file` — The TextFile class

This module provides the `TextFile` class, which gives an interface to text files that (optionally) takes care of stripping comments, ignoring blank lines, and joining lines with backslashes.

```python
class distutils.text_file.TextFile([filename=None, file=None, **options])
```

This class provides a file-like object that takes care of all the things you commonly want to do when processing a text file that has some line-by-line syntax: strip comments (as long as `#` is your comment character), skip blank lines, join adjacent lines by escaping the newline (ie. backslash at end of line), strip leading and/or trailing whitespace. All of these are optional and independently controllable.

The class provides a `warn()` method so you can generate warning messages that report physical line number, even if the logical line in question spans multiple physical lines. Also provides `unreadline()` for implementing line-at-a-time lookahead.

`TextFile` instances are create with either `filename`, `file`, or both. `RuntimeError` is raised if both are `None`. `filename` should be a string, and `file` a file object (or something that provides `readline()` and `close()` methods). It is recommended that you supply at least `filename`, so that `TextFile` can include it in warning messages. If `file` is not supplied, `TextFile` creates its own using the `open()` built-in function.
The options are all boolean, and affect the values returned by `readline()`

<table>
<thead>
<tr>
<th>option name</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strip_comments</td>
<td>strip from ' #' to end-of-line, as well as any whitespace leading up to the ' #'—unless it is escaped by a backslash</td>
<td>true</td>
</tr>
<tr>
<td>lstrip_ws</td>
<td>strip leading whitespace from each line before returning it</td>
<td>false</td>
</tr>
<tr>
<td>rstrip_ws</td>
<td>strip trailing whitespace (including line terminator!) from each line before returning it.</td>
<td>true</td>
</tr>
<tr>
<td>skip_blanks</td>
<td>skip lines that are empty <em>after</em> stripping comments and whitespace. (If both lstrip_ws and rstrip_ws are false, then some lines may consist of solely whitespace: these will <em>not</em> be skipped, even if skip_blanks is true.)</td>
<td>true</td>
</tr>
<tr>
<td>join_lines</td>
<td>if a backslash is the last non-newline character on a line after stripping comments and whitespace, join the following line to it to form one logical line; if N consecutive lines end with a backslash, then N+1 physical lines will be joined to form one logical line.</td>
<td>false</td>
</tr>
<tr>
<td>collapse_join</td>
<td>strip leading whitespace from lines that are joined to their predecessor; only matters if</td>
<td>false</td>
</tr>
</tbody>
</table>
Note that since rstrip_ws can strip the trailing newline, the semantics of `readline()` must differ from those of the built-in file object’s `readline()` method! In particular, `readline()` returns None for end-of-file: an empty string might just be a blank line (or an all-whitespace line), if rstrip_ws is true but skip_blanks is not.

`open(filename)`

Open a new file `filename`. This overrides any file or filename constructor arguments.

`close()`

Close the current file and forget everything we know about it (including the filename and the current line number).

`warn(msg[, line=None])`

Print (to stderr) a warning message tied to the current logical line in the current file. If the current logical line in the file spans multiple physical lines, the warning refers to the whole range, such as "lines 3-5". If `line` is supplied, it overrides the current line number; it may be a list or tuple to indicate a range of physical lines, or an integer for a single physical line.

`readline()`

Read and return a single logical line from the current file (or from an internal buffer if lines have previously been “unread” with `unreadline()`). If the `join_lines` option is true, this may involve reading multiple physical lines concatenated into a single string. Updates the current line number, so calling `warn()` after `readline()` emits a warning about the physical
line(s) just read. Returns None on end-of-file, since the empty string can occur if rstrip_ws is true but strip_blanks is not.

readlines()
Read and return the list of all logical lines remaining in the current file. This updates the current line number to the last line of the file.

unreadline(line)
Push line (a string) onto an internal buffer that will be checked by future readline() calls. Handy for implementing a parser with line-at-a-time lookahead. Note that lines that are “unread” with unreadline() are not subsequently re-cleansed (whitespace stripped, or whatever) when read with readline(). If multiple calls are made to unreadline() before a call to readline(), the lines will be returned most in most recent first order.
10.22. distutils.version — Version number classes
10.23. distutils.cmd — Abstract base class for Distutils commands

This module supplies the abstract base class Command.

class distutils.cmd.Command(dist)

Abstract base class for defining command classes, the “worker bees” of the Distutils. A useful analogy for command classes is to think of them as subroutines with local variables called options. The options are declared in initialize_options() and defined (given their final values) in finalize_options(), both of which must be defined by every command class. The distinction between the two is necessary because option values might come from the outside world (command line, config file, ...), and any options dependent on other options must be computed after these outside influences have been processed — hence finalize_options(). The body of the subroutine, where it does all its work based on the values of its options, is the run() method, which must also be implemented by every command class.

The class constructor takes a single argument dist, a Distribution instance.
10.24. Creating a new Distutils command

This section outlines the steps to create a new Distutils command.

A new command lives in a module in the `distutils.command` package. There is a sample template in that directory called `command_template`. Copy this file to a new module with the same name as the new command you’re implementing. This module should implement a class with the same name as the module (and the command). So, for instance, to create the command `peel_banana` (so that users can run `setup.py peel_banana`), you’d copy `command_template` to `distutils/command/peel_banana.py`, then edit it so that it’s implementing the class `peel_banana`, a subclass of `distutils.cmd.Command`.

Subclasses of `Command` must define the following methods.

**Command. initialize_options()**

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, `initialize_options()` implementations are just a bunch of `self.foo = None` assignments.

**Command. finalize_options()**

Set final values for all the options that this command supports. This is always called as late as possible, i.e. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if `foo` depends on `bar`, then it is safe to set `foo` from `bar` as long as
foo still has the same value it was assigned in initialize_options().

Command. **run()**

A command's raison d'être: carry out the action it exists to perform, controlled by the options initialized in initialize_options(), customized by other commands, the setup script, the command-line, and config files, and finalized in finalize_options(). All terminal output and filesystem interaction should be done by run().

Command. **sub_commands**

**sub_commands** formalizes the notion of a “family” of commands, e.g. install as the parent with sub-commands install_lib, install_headers, etc. The parent of a family of commands defines **sub_commands** as a class attribute; it's a list of 2-tuples (command_name, predicate), with command_name a string and predicate a function, a string or None. predicate is a method of the parent command that determines whether the corresponding command is applicable in the current situation. (E.g. install_headers is only applicable if we have any C header files to install.) If predicate is None, that command is always applicable.

**sub_commands** is usually defined at the end of a class, because predicates can be methods of the class, so they must already have been defined. The canonical example is the install command.
10.25. distutils.command —
Individual Distutils commands
10.26. `distutils.command.bdist` — Build a binary installer
10.27. 
**distutils.command.bdist_packager**  
— Abstract base class for packagers
10.28. 
\texttt{distutils.command.bdist_dumb} —
Build a “dumb” installer
10.29.
distutils.command.bdist_msi —
Build a Microsoft Installer binary package

class distutils.command.bdist_msi.bdist_msi

Builds a Windows Installer (.msi) binary package.

In most cases, the bdist_msi installer is a better choice than the bdist_wininst installer, because it provides better support for Win64 platforms, allows administrators to perform non-interactive installations, and allows installation through group policies.
10.30.
```
distutils.command.bdist_rpm —
Build a binary distribution as a Redhat RPM and SRPM
```
10.31.
`distutils.command.bdist_wininst` — Build a Windows installer
10.32. distutils.command.sdist — Build a source distribution
10.33. `distutils.command.build` —
Build all files of a package
10.34.
*distutils.command.build_clib* —
Build any C libraries in a package
10.35.
distutils.command.build_ext —
Build any extensions in a package
10.36. `distutils.command.build_py` — Build the .py/.pyc files of a package

`class distutils.command.build_py. build_py`

`class distutils.command.build_py. build_py_2to3`

Alternative implementation of build_py which also runs the 2to3 conversion library on each .py file that is going to be installed. To use this in a setup.py file for a distribution that is designed to run with both Python 2.x and 3.x, add:

```python
try:
    from distutils.command.build_py import build_py
except ImportError:
    from distutils.command.build_py import build_py_2to3

from distutils.command.build_py import build_py

try:
    from distutils.command.build_py import build_py
except ImportError:
    from distutils.command.build_py import build_py_2to3

from distutils.command.build_py import build_py
```

to your setup.py, and later:

```python
cmdclass = {'build_py': build_py}
```

to the invocation of setup().
10.37.  
distutils.command.build_scripts  
— Build the scripts of a package
10.38. `distutils.command.clean` — Clean a package build area

This command removes the temporary files created by `build` and its subcommands, like intermediary compiled object files. With the `--all` option, the complete build directory will be removed.

Extension modules built in place will not be cleaned, as they are not in the build directory.
10.39. distutils.command.config — Perform package configuration
10.40. `distutils.command.install` — Install a package
10.41. \texttt{distutils.command.install\_data} — Install data files from a package
10.42. `distutils.command.install_header` — Install C/C++ header files from a package.
10.43. 
\texttt{distutils.command.install\_lib} —
Install library files from a package
10.44.

distutils.command.install_script
— Install script files from a package
10.45. `distutils.command.register` — Register a module with the Python Package Index

The `register` command registers the package with the Python Package Index. This is described in more detail in PEP 301.
10.46. distutils.command.check — Check the meta-data of a package

The check command performs some tests on the meta-data of a package. For example, it verifies that all required meta-data are provided as the arguments passed to the `setup()` function.
Installing Python Modules (Legacy version)

**Author:** Greg Ward

**See also:**

**Installing Python Modules**

The up to date module installation documentations

This document describes the Python Distribution Utilities ("Distutils") from the end-user’s point-of-view, describing how to extend the capabilities of a standard Python installation by building and installing third-party Python modules and extensions.

**Note:** This guide only covers the basic tools for building and distributing extensions that are provided as part of this version of Python. Third party tools offer easier to use and more secure alternatives. Refer to the quick recommendations section in the Python Packaging User Guide for more information.
Introduction

Although Python’s extensive standard library covers many programming needs, there often comes a time when you need to add some new functionality to your Python installation in the form of third-party modules. This might be necessary to support your own programming, or to support an application that you want to use and that happens to be written in Python.

In the past, there has been little support for adding third-party modules to an existing Python installation. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 2.0, this changed.

This document is aimed primarily at the people who need to install third-party Python modules: end-users and system administrators who just need to get some Python application running, and existing Python programmers who want to add some new goodies to their toolbox. You don’t need to know Python to read this document; there will be some brief forays into using Python’s interactive mode to explore your installation, but that’s it. If you’re looking for information on how to distribute your own Python modules so that others may use them, see the Distributing Python Modules (Legacy version) manual. Debugging the setup script may also be of interest.

Best case: trivial installation

In the best case, someone will have prepared a special version of the module distribution you want to install that is targeted specifically at your platform and is installed just like any other software on your platform. For example, the module developer might make an executable installer available for Windows users, an RPM package for
users of RPM-based Linux systems (Red Hat, SuSE, Mandrake, and many others), a Debian package for users of Debian-based Linux systems, and so forth.

In that case, you would download the installer appropriate to your platform and do the obvious thing with it: run it if it’s an executable installer, `rpm --install` it if it’s an RPM, etc. You don’t need to run Python or a setup script, you don’t need to compile anything—you might not even need to read any instructions (although it’s always a good idea to do so anyway).

Of course, things will not always be that easy. You might be interested in a module distribution that doesn’t have an easy-to-use installer for your platform. In that case, you’ll have to start with the source distribution released by the module’s author/maintainer. Installing from a source distribution is not too hard, as long as the modules are packaged in the standard way. The bulk of this document is about building and installing modules from standard source distributions.

**The new standard: Distutils**

If you download a module source distribution, you can tell pretty quickly if it was packaged and distributed in the standard way, i.e. using the Distutils. First, the distribution’s name and version number will be featured prominently in the name of the downloaded archive, e.g. `foo-1.0.tar.gz` or `widget-0.9.7.zip`. Next, the archive will unpack into a similarly-named directory: `foo-1.0` or `widget-0.9.7`. Additionally, the distribution will contain a setup script `setup.py`, and a file named `README.txt` or possibly just `README`, which should explain that building and installing the module distribution is a simple matter of running one command from a terminal:
python setup.py install

For Windows, this command should be run from a command prompt window (Start ▶ Accessories):

setup.py install

If all these things are true, then you already know how to build and install the modules you’ve just downloaded: Run the command above. Unless you need to install things in a non-standard way or customize the build process, you don’t really need this manual. Or rather, the above command is everything you need to get out of this manual.
Standard Build and Install

As described in section The new standard: Distutils, building and installing a module distribution using the Distutils is usually one simple command to run from a terminal:

```python
python setup.py install
```

Platform variations

You should always run the setup command from the distribution root directory, i.e. the top-level subdirectory that the module source distribution unpacks into. For example, if you’ve just downloaded a module source distribution foo-1.0.tar.gz onto a Unix system, the normal thing to do is:

```bash
gunzip -c foo-1.0.tar.gz | tar xf -   # unpacks into directory foo-1.0
cd foo-1.0
python setup.py install
```

On Windows, you’d probably download foo-1.0.zip. If you downloaded the archive file to C:\Temp, then it would unpack into C:\Temp\foo-1.0; you can use either an archive manipulator with a graphical user interface (such as WinZip) or a command-line tool (such as unzip or pkunzip) to unpack the archive. Then, open a command prompt window and run:

```bash
cd c:\Temp\foo-1.0
python setup.py install
```
Splitting the job up

Running `setup.py install` builds and installs all modules in one run. If you prefer to work incrementally—especially useful if you want to customize the build process, or if things are going wrong—you can use the setup script to do one thing at a time. This is particularly helpful when the build and install will be done by different users—for example, you might want to build a module distribution and hand it off to a system administrator for installation (or do it yourself, with super-user privileges).

For example, you can build everything in one step, and then install everything in a second step, by invoking the setup script twice:

```
python setup.py build
python setup.py install
```

If you do this, you will notice that running the `install` command first runs the `build` command, which—in this case—quickly notices that it has nothing to do, since everything in the `build` directory is up-to-date.

You may not need this ability to break things down often if all you do is install modules downloaded off the ‘net, but it’s very handy for more advanced tasks. If you get into distributing your own Python modules and extensions, you’ll run lots of individual Distutils commands on their own.

How building works

As implied above, the `build` command is responsible for putting the files to install into a `build directory`. By default, this is `build` under the
distribution root; if you’re excessively concerned with speed, or want to keep the source tree pristine, you can change the build directory with the --build-base option. For example:

```
python setup.py build --build-base=/path/to/pybuild/
```

(Or you could do this permanently with a directive in your system or personal Distutils configuration file; see section Distutils Configuration Files.) Normally, this isn’t necessary.

The default layout for the build tree is as follows:

```
--- build/  --- lib/
or
--- build/  --- lib.<plat>/
  temp.<plat>/
```

where `<plat>` expands to a brief description of the current OS/hardware platform and Python version. The first form, with just a `lib` directory, is used for “pure module distributions”—that is, module distributions that include only pure Python modules. If a module distribution contains any extensions (modules written in C/C++), then the second form, with two `<plat>` directories, is used. In that case, the `temp.<plat>` directory holds temporary files generated by the compile/link process that don’t actually get installed. In either case, the `lib` (or `lib.<plat>`) directory contains all Python modules (pure Python and extensions) that will be installed.

In the future, more directories will be added to handle Python scripts, documentation, binary executables, and whatever else is needed to handle the job of installing Python modules and applications.
How installation works

After the `build` command runs (whether you run it explicitly, or the `install` command does it for you), the work of the `install` command is relatively simple: all it has to do is copy everything under `build/lib` (or `build/lib.plat`) to your chosen installation directory.

If you don’t choose an installation directory—i.e., if you just run `setup.py install`—then the `install` command installs to the standard location for third-party Python modules. This location varies by platform and by how you built/installed Python itself. On Unix (and Mac OS X, which is also Unix-based), it also depends on whether the module distribution being installed is pure Python or contains extensions (“non-pure”):

<table>
<thead>
<tr>
<th>Platform</th>
<th>Standard installation location</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix (pure)</td>
<td><code>prefix/lib/pythonX.Y/site-packages</code></td>
<td><code>/usr/local/lib/packages</code></td>
</tr>
<tr>
<td>Unix (non-pure)</td>
<td><code>exec-prefix/lib/pythonX.Y/site-packages</code></td>
<td><code>/usr/local/lib/packages</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>prefix\Lib\site-packages</code></td>
<td><code>C:\PythonXY\Lib</code></td>
</tr>
</tbody>
</table>

Notes:

1. Most Linux distributions include Python as a standard part of the system, so `prefix` and `exec-prefix` are usually both `/usr` on Linux. If you build Python yourself on Linux (or any Unix-like system), the default `prefix` and `exec-prefix` are `/usr/local`.
2. The default installation directory on Windows was `C:\Program`
Files\Python under Python 1.6a1, 1.5.2, and earlier.

`prefix` and `exec-prefix` stand for the directories that Python is installed to, and where it finds its libraries at run-time. They are always the same under Windows, and very often the same under Unix and Mac OS X. You can find out what your Python installation uses for `prefix` and `exec-prefix` by running Python in interactive mode and typing a few simple commands. Under Unix, just type `python` at the shell prompt. Under Windows, choose Start ▶ Programs ▶ Python X.Y ▶ Python (command line). Once the interpreter is started, you type Python code at the prompt. For example, on my Linux system, I type the three Python statements shown below, and get the output as shown, to find out my `prefix` and `exec-prefix`:

```
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.exec_prefix
'/usr'
```

A few other placeholders are used in this document: `X.Y` stands for the version of Python, for example 3.2; `abiflags` will be replaced by the value of `sys.abiflags` or the empty string for platforms which don’t define ABI flags; `distname` will be replaced by the name of the module distribution being installed. Dots and capitalization are important in the paths; for example, a value that uses `python3.2` on UNIX will typically use `Python32` on Windows.

If you don’t want to install modules to the standard location, or if you don’t have permission to write there, then you need to read about
alternate installations in section *Alternate Installation*. If you want to customize your installation directories more heavily, see section *Custom Installation* on custom installations.
Alternate Installation

Often, it is necessary or desirable to install modules to a location other than the standard location for third-party Python modules. For example, on a Unix system you might not have permission to write to the standard third-party module directory. Or you might wish to try out a module before making it a standard part of your local Python installation. This is especially true when upgrading a distribution already present: you want to make sure your existing base of scripts still works with the new version before actually upgrading.

The Distutils install command is designed to make installing module distributions to an alternate location simple and painless. The basic idea is that you supply a base directory for the installation, and the install command picks a set of directories (called an installation scheme) under this base directory in which to install files. The details differ across platforms, so read whichever of the following sections applies to you.

Note that the various alternate installation schemes are mutually exclusive: you can pass --user, or --home, or --prefix and --exec-prefix, or --install-base and --install-platbase, but you can’t mix from these groups.

Alternate installation: the user scheme

This scheme is designed to be the most convenient solution for users that don’t have write permission to the global site-packages directory or don’t want to install into it. It is enabled with a simple option:

```
python setup.py install --user
```
Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`). Here are the values for UNIX, including Mac OS X:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase/bin</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase/include/pythonX.Yabiflags/distname</code></td>
</tr>
</tbody>
</table>

And here are the values used on Windows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase\PythonXY\site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase\PythonXY\Scripts</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase\PythonXY\Include{distname}</code></td>
</tr>
</tbody>
</table>

The advantage of using this scheme compared to the other ones described below is that the user site-packages directory is under normal conditions always included in `sys.path` (see `site` for more information), which means that there is no additional step to perform after running the `setup.py` script to finalize the installation.

The `build_ext` command also has a `--user` option to add `userbase/include` to the compiler search path for header files and
*userbase*/lib to the compiler search path for libraries as well as to the runtime search path for shared C libraries (rpath).

Alternate installation: the home scheme

The idea behind the “home scheme” is that you build and maintain a personal stash of Python modules. This scheme’s name is derived from the idea of a “home” directory on Unix, since it’s not unusual for a Unix user to make their home directory have a layout similar to */usr/* or */usr/local/*. This scheme can be used by anyone, regardless of the operating system they are installing for.

Installing a new module distribution is as simple as

```
python setup.py install --home=<dir>
```

where you can supply any directory you like for the `--home` option. On Unix, lazy typists can just type a tilde (~); the `install` command will expand this to your home directory:

```
python setup.py install --home=~
```

To make Python find the distributions installed with this scheme, you may have to modify Python’s search path or edit `sitecustomize` (see `site`) to call `site.addsitedir()` or edit `sys.path`.

The `--home` option defines the installation base directory. Files are installed to the following directories under the installation base as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><em>home</em>/lib/python</td>
</tr>
</tbody>
</table>
scripts | home/bin
---|---
data | home
C headers | home/include/python/distname

(Mentally replace slashes with backslashes if you’re on Windows.)

**Alternate installation: Unix (the prefix scheme)**

The “prefix scheme” is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that’s why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of “the system” rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.X` rather than `/usr/lib/python2.X`. This can be done with

```
/usr/bin/python setup.py install --prefix=/usr/local
```

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.X`, but those modules would have to be
installed to, say, /mnt/@server/export/lib/python2.X. This could be done with

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

In either case, the `--prefix` option defines the installation base, and the `--exec-prefix` option defines the platform-specific installation base, which is used for platform-specific files. (Currently, this just means non-pure module distributions, but could be expanded to C libraries, binary executables, etc.) If `--exec-prefix` is not supplied, it defaults to `--prefix`. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td><code>prefix/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>extension modules</td>
<td><code>exec-prefix/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>prefix/bin</code></td>
</tr>
<tr>
<td>data</td>
<td><code>prefix</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>prefix/include/pythonX.Y ABIFlags/distname</code></td>
</tr>
</tbody>
</table>

There is no requirement that `--prefix` or `--exec-prefix` actually point to an alternate Python installation; if the directories listed above do not already exist, they are created at installation time.

Incidentally, the real reason the prefix scheme is important is simply that a standard Unix installation uses the prefix scheme, but with `--prefix` and `--exec-prefix` supplied by Python itself as
sys.prefix and sys.exec_prefix. Thus, you might think you'll never use the prefix scheme, but every time you run python setup.py install without any other options, you're using it.

Note that installing extensions to an alternate Python installation has no effect on how those extensions are built: in particular, the Python header files (Python.h and friends) installed with the Python interpreter used to run the setup script will be used in compiling extensions. It is your responsibility to ensure that the interpreter used to run extensions installed in this way is compatible with the interpreter used to build them. The best way to do this is to ensure that the two interpreters are the same version of Python (possibly different builds, or possibly copies of the same build). (Of course, if your --prefix and --exec-prefix don't even point to an alternate Python installation, this is immaterial.)

Alternate installation: Windows (the prefix scheme)

Windows has no concept of a user's home directory, and since the standard Python installation under Windows is simpler than under Unix, the --prefix option has traditionally been used to install additional packages in separate locations on Windows.

```
python setup.py install --prefix="\Temp\Python"
```

to install modules to the Temp\Python directory on the current drive.

The installation base is defined by the --prefix option; the --exec-prefix option is not supported under Windows, which means
that pure Python modules and extension modules are installed into the same location. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>\texttt{prefix\Lib\site-packages}</td>
</tr>
<tr>
<td>scripts</td>
<td>\texttt{prefix\Scripts}</td>
</tr>
<tr>
<td>data</td>
<td>\texttt{prefix}</td>
</tr>
<tr>
<td>C headers</td>
<td>\texttt{prefix\Include{distname}}</td>
</tr>
</tbody>
</table>
Custom Installation

Sometimes, the alternate installation schemes described in section Alternate Installation just don’t do what you want. You might want to tweak just one or two directories while keeping everything under the same base directory, or you might want to completely redefine the installation scheme. In either case, you’re creating a custom installation scheme.

To create a custom installation scheme, you start with one of the alternate schemes and override some of the installation directories used for the various types of files, using these options:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>--install-purelib</td>
</tr>
<tr>
<td>extension modules</td>
<td>--install-platlib</td>
</tr>
<tr>
<td>all modules</td>
<td>--install-lib</td>
</tr>
<tr>
<td>scripts</td>
<td>--install-scripts</td>
</tr>
<tr>
<td>data</td>
<td>--install-data</td>
</tr>
<tr>
<td>C headers</td>
<td>--install-headers</td>
</tr>
</tbody>
</table>

These override options can be relative, absolute, or explicitly defined in terms of one of the installation base directories. (There are two installation base directories, and they are normally the same—they only differ when you use the Unix “prefix scheme” and supply different --prefix and --exec-prefix options; using --install-lib
will override values computed or given for `--install-purelib` and `--install-platlib`, and is recommended for schemes that don’t make a difference between Python and extension modules.)

For example, say you’re installing a module distribution to your home directory under Unix—but you want scripts to go in `~/scripts` rather than `~/bin`. As you might expect, you can override this directory with the `--install-scripts` option; in this case, it makes most sense to supply a relative path, which will be interpreted relative to the installation base directory (your home directory, in this case):

```
python setup.py install --home=~ --install-scripts=scripts
```

Another Unix example: suppose your Python installation was built and installed with a prefix of `/usr/local/python`, so under a standard installation scripts will wind up in `/usr/local/python/bin`. If you want them in `/usr/local/bin` instead, you would supply this absolute directory for the `--install-scripts` option:

```
python setup.py install --install-scripts=/usr/local/bin
```

(This performs an installation using the “prefix scheme,” where the prefix is whatever your Python interpreter was installed with—`/usr/local/python` in this case.)

If you maintain Python on Windows, you might want third-party modules to live in a subdirectory of `prefix`, rather than right in `prefix` itself. This is almost as easy as customizing the script installation directory—you just have to remember that there are two types of modules to worry about, Python and extension modules, which can conveniently be both controlled by one option:
The specified installation directory is relative to `prefix`. Of course, you also have to ensure that this directory is in Python's module search path, such as by putting a `.pth` file in a site directory (see `site`). See section `Modifying Python's Search Path` to find out how to modify Python's search path.

If you want to define an entire installation scheme, you just have to supply all of the installation directory options. The recommended way to do this is to supply relative paths; for example, if you want to maintain all Python module-related files under `python` in your home directory, and you want a separate directory for each platform that you use your home directory from, you might define the following installation scheme:

```bash
python setup.py install --home=~ \
   --install-purelib=python/lib \
   --install-platlib=python/lib \
   --install-scripts=python/scripts \
   --install-data=python/data
```

or, equivalently,

```bash
python setup.py install --home=~/.python \
   --install-purelib=lib \
   --install-platlib='lib.$PLAT' \
   --install-scripts=scripts \
   --install-data=data
```

 `$PLAT` is not (necessarily) an environment variable—it will be expanded by the Distutils as it parses your command line options, just
as it does when parsing your configuration file(s).

Obviously, specifying the entire installation scheme every time you install a new module distribution would be very tedious. Thus, you can put these options into your Distutils config file (see section Distutils Configuration Files):

```
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
install-data=python/data
```

or, equivalently,

```
[install]
install-base=$HOME/python
install-purelib=lib
install-platlib=lib.$PLAT
install-scripts=scripts
install-data=data
```

Note that these two are not equivalent if you supply a different installation base directory when you run the setup script. For example,

```
python setup.py install --install-base=/tmp
```

would install pure modules to /tmp/python/lib in the first case, and to /tmp/lib in the second case. (For the second case, you probably want to supply an installation base of /tmp/python.)

You probably noticed the use of $HOME and $PLAT in the sample configuration file input. These are Distutils configuration variables,
which bear a strong resemblance to environment variables. In fact, you can use environment variables in config files on platforms that have such a notion but the Distutils additionally define a few extra variables that may not be in your environment, such as $PLAT$. (And of course, on systems that don’t have environment variables, such as Mac OS 9, the configuration variables supplied by the Distutils are the only ones you can use.) See section Distutils Configuration Files for details.

**Note:** When a virtual environment is activated, any options that change the installation path will be ignored from all distutils configuration files to prevent inadvertently installing projects outside of the virtual environment.

### Modifying Python’s Search Path

When the Python interpreter executes an `import` statement, it searches for both Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the `sys` module and printing the value of `sys.path`.

```bash
$ python
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> import sys
>>> sys.path
```
The null string in `sys.path` represents the current working directory.

The expected convention for locally installed packages is to put them in the `../site-packages/` directory, but you may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server under `/www`. Add-on Python modules might then belong in `/www/python`, and in order to import them, this directory must be added to `sys.path`. There are several different ways to add the directory.

The most convenient way is to add a path configuration file to a directory that's already on Python's path, usually to the `../site-packages/` directory. Path configuration files have an extension of `.pth`, and each line must contain a single path that will be appended to `sys.path`. (Because the new paths are appended to `sys.path`, modules in the added directories will not override standard modules. This means you can't use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they're relative to the directory containing the `.pth` file. See the documentation of the `site` module for more information.

A slightly less convenient way is to edit the `site.py` file in Python's standard library, and modify `sys.path`. `site.py` is automatically imported when the Python interpreter is executed, unless the `-S` switch is supplied to suppress this behaviour. So you could simply edit `site.py` and add two lines to it:

```python
import sys
sys.path.append( '/www/python/' )
```
However, if you reinstall the same major version of Python (perhaps when upgrading from 2.2 to 2.2.2, for example) site.py will be overwritten by the stock version. You’d have to remember that it was modified and save a copy before doing the installation.

There are two environment variables that can modify sys.path. PYTHONHOME sets an alternate value for the prefix of the Python installation. For example, if PYTHONHOME is set to /www/python, the search path will be set to [', '/www/python/lib/pythonX.Y/',' '/www/python/lib/pythonX.Y/plat-linux2', ...].

The PYTHONPATH variable can be set to a list of paths that will be added to the beginning of sys.path. For example, if PYTHONPATH is set to /www/python:/opt/py, the search path will begin with ['/www/python', '/opt/py']. (Note that directories must exist in order to be added to sys.path; the site module removes paths that don't exist.)

Finally, sys.path is just a regular Python list, so any Python application can modify it by adding or removing entries.
Distutils Configuration Files

As mentioned above, you can use Distutils configuration files to record personal or site preferences for any Distutils options. That is, any option to any command can be stored in one of two or three (depending on your platform) configuration files, which will be consulted before the command-line is parsed. This means that configuration files will override default values, and the command-line will in turn override configuration files. Furthermore, if multiple configuration files apply, values from “earlier” files are overridden by “later” files.

Location and names of config files

The names and locations of the configuration files vary slightly across platforms. On Unix and Mac OS X, the three configuration files (in the order they are processed) are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td><code>prefix/lib/pythonver/distutils/distutils.cfg</code></td>
</tr>
<tr>
<td>personal</td>
<td><code>$HOME/.pydistutils.cfg</code></td>
</tr>
<tr>
<td>local</td>
<td><code>setup.cfg</code></td>
</tr>
</tbody>
</table>

And on Windows, the configuration files are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td><code>prefix\Lib\distutils\distutils.cfg</code></td>
<td>(4)</td>
</tr>
<tr>
<td>personal</td>
<td><code>%HOME%\pydistutils.cfg</code></td>
<td>(5)</td>
</tr>
</tbody>
</table>
On all platforms, the “personal” file can be temporarily disabled by passing the `--no-user-cfg` option.

Notes:

1. Strictly speaking, the system-wide configuration file lives in the directory where the Distutils are installed; under Python 1.6 and later on Unix, this is as shown. For Python 1.5.2, the Distutils will normally be installed to `prefix/lib/python1.5/site-packages/distutils`, so the system configuration file should be put there under Python 1.5.2.

2. On Unix, if the `HOME` environment variable is not defined, the user’s home directory will be determined with the `getpwuid()` function from the standard `pwd` module. This is done by the `os.path.expanduser()` function used by Distutils.

3. I.e., in the current directory (usually the location of the setup script).

4. (See also note (1).) Under Python 1.6 and later, Python’s default “installation prefix” is `C:\Python`, so the system configuration file is normally `C:\Python\Lib\distutils\distutils.cfg`. Under Python 1.5.2, the default prefix was `C:\Program Files\Python`, and the Distutils were not part of the standard library—so the system configuration file would be `C:\Program Files\Python\distutils\distutils.cfg` in a standard Python 1.5.2 installation under Windows.

5. On Windows, if the `HOME` environment variable is not defined, `USERPROFILE` then `HOMEDRIVE` and `HOMEPATH` will be tried. This is done by the `os.path.expanduser()` function used by Distutils.

Syntax of config files
The Distutils configuration files all have the same syntax. The config files are grouped into sections. There is one section for each Distutils command, plus a global section for global options that affect every command. Each section consists of one option per line, specified as option=value.

For example, the following is a complete config file that just forces all commands to run quietly by default:

```
[global]
verbose=0
```

If this is installed as the system config file, it will affect all processing of any Python module distribution by any user on the current system. If it is installed as your personal config file (on systems that support them), it will affect only module distributions processed by you. And if it is used as the setup.cfg for a particular module distribution, it affects only that distribution.

You could override the default “build base” directory and make the build* commands always forcibly rebuild all files with the following:

```
[build]
build-base=blib
force=1
```

which corresponds to the command-line arguments

```
python setup.py build --build-base=blib --force
```

except that including the build command on the command-line means that command will be run. Including a particular command in config
files has no such implication; it only means that if the command is run, the options in the config file will apply. (Or if other commands that derive values from it are run, they will use the values in the config file.)

You can find out the complete list of options for any command using the --help option, e.g.:

```python
python setup.py build --help
```

and you can find out the complete list of global options by using --help without a command:

```python
python setup.py --help
```

See also the “Reference” section of the “Distributing Python Modules” manual.
Building Extensions: Tips and Tricks

Whenever possible, the Distutils try to use the configuration information made available by the Python interpreter used to run the setup.py script. For example, the same compiler and linker flags used to compile Python will also be used for compiling extensions. Usually this will work well, but in complicated situations this might be inappropriate. This section discusses how to override the usual Distutils behaviour.

Tweaking compiler/linker flags

Compiling a Python extension written in C or C++ will sometimes require specifying custom flags for the compiler and linker in order to use a particular library or produce a special kind of object code. This is especially true if the extension hasn't been tested on your platform, or if you're trying to cross-compile Python.

In the most general case, the extension author might have foreseen that compiling the extensions would be complicated, and provided a Setup file for you to edit. This will likely only be done if the module distribution contains many separate extension modules, or if they often require elaborate sets of compiler flags in order to work.

A Setup file, if present, is parsed in order to get a list of extensions to build. Each line in a Setup describes a single module. Lines have the following structure:

```
module ... [sourcefile ...] [cpparg ...] [library ...
```

Let's examine each of the fields in turn.
- **module** is the name of the extension module to be built, and should be a valid Python identifier. You can't just change this in order to rename a module (edits to the source code would also be needed), so this should be left alone.

- **sourcefile** is anything that's likely to be a source code file, at least judging by the filename. Filenames ending in `.c` are assumed to be written in C, filenames ending in `.C`, `.cc`, and `.c++` are assumed to be C++, and filenames ending in `.m` or `.mm` are assumed to be in Objective C.

- **cpparg** is an argument for the C preprocessor, and is anything starting with `-I`, `-D`, `-U` or `-C`.

- **library** is anything ending in `.a` or beginning with `-l` or `-L`.

If a particular platform requires a special library on your platform, you can add it by editing the `Setup` file and running `python setup.py build`. For example, if the module defined by the line

```
foo foomodule.c
```

must be linked with the math library `libm.a` on your platform, simply add `-lm` to the line:

```
foo foomodule.c -lm
```

Arbitrary switches intended for the compiler or the linker can be supplied with the `-Xcompiler arg` and `-Xlinker arg` options:

```
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```

The next option after `-Xcompiler` and `-Xlinker` will be appended to the proper command line, so in the above example the compiler will be passed the `-o32` option, and the linker will be passed `-shared`. If
a compiler option requires an argument, you’ll have to supply multiple -Xcompiler options; for example, to pass -x c++ the Setup file would have to contain -Xcompiler -x -Xcompiler c++.

Compiler flags can also be supplied through setting the CFLAGS environment variable. If set, the contents of CFLAGS will be added to the compiler flags specified in the Setup file.

Using non-Microsoft compilers on Windows
Borland/CodeGear C++

This subsection describes the necessary steps to use Distutils with the Borland C++ compiler version 5.5. First you have to know that Borland’s object file format (OMF) is different from the format used by the Python version you can download from the Python or ActiveState Web site. (Python is built with Microsoft Visual C++, which uses COFF as the object file format.) For this reason you have to convert Python’s library python25.lib into the Borland format. You can do this as follows:

```
coff2omf python25.lib python25_bcpp.lib
```

The coff2omf program comes with the Borland compiler. The file python25.lib is in the Libs directory of your Python installation. If your extension uses other libraries (zlib, ...) you have to convert them too.

The converted files have to reside in the same directories as the normal libraries.

How does Distutils manage to use these libraries with their changed
names? If the extension needs a library (eg. foo) Distutils checks first if it finds a library with suffix _bcpp (eg. foo_bcpp.lib) and then uses this library. In the case it doesn’t find such a special library it uses the default name (foo.lib.) [1]

To let Distutils compile your extension with Borland C++ you now have to type:

```
python setup.py build --compiler=bcpp
```

If you want to use the Borland C++ compiler as the default, you could specify this in your personal or system-wide configuration file for Distutils (see section Distutils Configuration Files.)

See also:

**C++Builder Compiler**
Information about the free C++ compiler from Borland, including links to the download pages.

**Creating Python Extensions Using Borland’s Free Compiler**
Document describing how to use Borland’s free command-line C++ compiler to build Python.

**GNU C / Cygwin / MinGW**

This section describes the necessary steps to use Distutils with the GNU C/C++ compilers in their Cygwin and MinGW distributions. [2]
For a Python interpreter that was built with Cygwin, everything should work without any of these following steps.

Not all extensions can be built with MinGW or Cygwin, but many can. Extensions most likely to not work are those that use C++ or depend
on Microsoft Visual C extensions.

To let Distutils compile your extension with Cygwin you have to type:

```
python setup.py build --compiler=cygwin
```

and for Cygwin in no-cygwin mode [3] or for MinGW type:

```
python setup.py build --compiler=mingw32
```

If you want to use any of these options/compilers as default, you should consider writing it in your personal or system-wide configuration file for Distutils (see section Distutils Configuration Files.)

Older Versions of Python and MinGW

The following instructions only apply if you’re using a version of Python inferior to 2.4.1 with a MinGW inferior to 3.0.0 (with binutils-2.13.90-20030111-1).

These compilers require some special libraries. This task is more complex than for Borland's C++, because there is no program to convert the library. First you have to create a list of symbols which the Python DLL exports. (You can find a good program for this task at https://sourceforge.net/projects/mingw/files/MinGW/Extension/pexports/)

```
pexports python25.dll >python25.def
```

The location of an installed python25.dll will depend on the installation options and the version and language of Windows. In a “just for me” installation, it will appear in the root of the installation directory. In a shared installation, it will be located in the system directory.
Then you can create from these information an import library for gcc.

```
/cygwin/bin/dlltool --dllname python25.dll --def python25.def --output-lib libpython25.a
```

The resulting library has to be placed in the same directory as `python25.lib`. (Should be the `libs` directory under your Python installation directory.)

If your extension uses other libraries (zlib,...) you might have to convert them too. The converted files have to reside in the same directories as the normal libraries do.

See also:

**Building Python modules on MS Windows platform with MinGW**

Information about building the required libraries for the MinGW environment.

Footnotes

[1] This also means you could replace all existing COFF-libraries with OMF-libraries of the same name.


[3] Then you have no POSIX emulation available, but you also don't need `cygwin1.dll`. 
Distributing Python Modules
(Legacy version)

Authors: Greg Ward, Anthony Baxter
Email: distutils-sig@python.org

See also:

**Distributing Python Modules**
   The up to date module distribution documentations

This document describes the Python Distribution Utilities ("Distutils") from the module developer's point of view, describing how to use the Distutils to make Python modules and extensions easily available to a wider audience with very little overhead for build/release/install mechanics.

**Note:** This guide only covers the basic tools for building and distributing extensions that are provided as part of this version of Python. Third party tools offer easier to use and more secure alternatives. Refer to the quick recommendations section in the Python Packaging User Guide for more information.

- **1. An Introduction to Distutils**
  - **1.1. Concepts & Terminology**
  - **1.2. A Simple Example**
  - **1.3. General Python terminology**
  - **1.4. Distutils-specific terminology**
- **2. Writing the Setup Script**
  - **2.1. Listing whole packages**
2.2. Listing individual modules
2.3. Describing extension modules
2.4. Relationships between Distributions and Packages
2.5. Installing Scripts
2.6. Installing Package Data
2.7. Installing Additional Files
2.8. Additional meta-data
2.9. Debugging the setup script

3. Writing the Setup Configuration File

4. Creating a Source Distribution
   4.1. Specifying the files to distribute
   4.2. Manifest-related options

5. Creating Built Distributions
   5.1. Creating RPM packages
   5.2. Creating Windows Installers
   5.3. Cross-compiling on Windows
   5.4. Vista User Access Control (UAC)

6. The Python Package Index (PyPI)
   6.1. PyPI overview
   6.2. Distutils commands
   6.3. PyPI package display

7. Examples
   7.1. Pure Python distribution (by module)
   7.2. Pure Python distribution (by package)
   7.3. Single extension module
   7.4. Checking a package
   7.5. Reading the metadata

8. Extending Distutils
   8.1. Integrating new commands
   8.2. Adding new distribution types

9. Command Reference
   9.1. Installing modules: the install command family
   9.2. Creating a source distribution: the sdist command

10. API Reference
10.1. distutils.core — Core Distutils functionality
10.2. distutils.ccompiler — CCompiler base class
10.3. distutils.unixccompiler — Unix C Compiler
10.4. distutils.msvccompiler — Microsoft Compiler
10.5. distutils.bcppcompiler — Borland Compiler
10.6. distutils.cygwincompiler — Cygwin Compiler
10.7. distutils.archive_util — Archiving utilities
10.8. distutils.dep_util — Dependency checking
10.9. distutils.dir_util — Directory tree operations
10.10. distutils.file_util — Single file operations
10.11. distutils.util — Miscellaneous other utility functions
10.12. distutils.dist — The Distribution class
10.13. distutils.extension — The Extension class
10.14. distutils.debug — Distutils debug mode
10.15. distutils.errors — Distutils exceptions
10.16. distutils.fancy_getopt — Wrapper around the standard getopt module
10.17. distutils.filelist — The FileList class
10.18. distutils.log — Simple PEP 282-style logging
10.19. distutils.spawn — Spawn a sub-process
10.20. distutils.sysconfig — System configuration information
10.21. distutils.text_file — The TextFile class
10.22. distutils.version — Version number classes
10.23. distutils.cmd — Abstract base class for Distutils commands
10.24. Creating a new Distutils command
10.25. distutils.command — Individual Distutils commands
10.26. distutils.command.bdist — Build a binary installer
- distutils.command.bdist_packager — Abstract base class for packagers
- distutils.command.bdist_dumb — Build a “dumb” installer
- distutils.command.bdist_msi — Build a Microsoft Installer binary package
- distutils.command.bdist_rpm — Build a binary distribution as a Redhat RPM and SRPM
- distutils.command.bdist_wininst — Build a Windows installer
- distutils.command.sdist — Build a source distribution
- distutils.command.build — Build all files of a package
- distutils.command.build_clib — Build any C libraries in a package
- distutils.command.build_ext — Build any extensions in a package
- distutils.command.build_py — Build the .py/.pyc files of a package
- distutils.command.build_scripts — Build the scripts of a package
- distutils.command.clean — Clean a package build area
- distutils.command.config — Perform package configuration
- distutils.command.install — Install a package
- distutils.command.install_data — Install data files from a package
- distutils.command.install_headers — Install C/C++ header files from a package
- distutils.command.install_lib — Install library files from a package
- 10.44. distutils.command.install_scripts — Install script files from a package
- 10.45. distutils.command.register — Register a module with the Python Package Index
- 10.46. distutils.command.check — Check the metadata of a package
8. Extending Distutils

Distutils can be extended in various ways. Most extensions take the form of new commands or replacements for existing commands. New commands may be written to support new types of platform-specific packaging, for example, while replacements for existing commands may be made to modify details of how the command operates on a package.

Most extensions of the distutils are made within setup.py scripts that want to modify existing commands; many simply add a few file extensions that should be copied into packages in addition to .py files as a convenience.

Most distutils command implementations are subclasses of the distutils.cmd.Command class. New commands may directly inherit from Command, while replacements often derive from Command indirectly, directly subclassing the command they are replacing. Commands are required to derive from Command.
8.1. Integrating new commands

There are different ways to integrate new command implementations into distutils. The most difficult is to lobby for the inclusion of the new features in distutils itself, and wait for (and require) a version of Python that provides that support. This is really hard for many reasons.

The most common, and possibly the most reasonable for most needs, is to include the new implementations with your setup.py script, and cause the `distutils.core.setup()` function use them:

```python
from distutils.command.build_py import build_py as _build_py
from distutils.core import setup

class build_py(_build_py):
    """Specialized Python source builder.""

    # implement whatever needs to be different...

setup(cmdclass={'build_py': build_py},
     ...
```

This approach is most valuable if the new implementations must be used to use a particular package, as everyone interested in the package will need to have the new command implementation.

Beginning with Python 2.4, a third option is available, intended to allow new commands to be added which can support existing `setup.py` scripts without requiring modifications to the Python installation. This is expected to allow third-party extensions to provide support for additional packaging systems, but the commands can be used for anything distutils commands can be used for. A new configuration
option, command_packages (command-line option --command-packages), can be used to specify additional packages to be searched for modules implementing commands. Like all distutils options, this can be specified on the command line or in a configuration file. This option can only be set in the [global] section of a configuration file, or before any commands on the command line. If set in a configuration file, it can be overridden from the command line; setting it to an empty string on the command line causes the default to be used. This should never be set in a configuration file provided with a package.

This new option can be used to add any number of packages to the list of packages searched for command implementations; multiple package names should be separated by commas. When not specified, the search is only performed in the distutils.command package. When setup.py is run with the option --command-packages distcmds,buildcmds, however, the packages distutils.command, distcmds, and buildcmds will be searched in that order. New commands are expected to be implemented in modules of the same name as the command by classes sharing the same name. Given the example command line option above, the command bdist_openpkg could be implemented by the class distcmds.bdist_openpkg.bdist_openpkg or buildcmds.bdist_openpkg.bdist_openpkg.
8.2. Adding new distribution types

Commands that create distributions (files in the dist/ directory) need to add (command, filename) pairs to self.distribution.dist_files so that upload can upload it to PyPI. The filename in the pair contains no path information, only the name of the file itself. In dry-run mode, pairs should still be added to represent what would have been created.
4. Creating a Source Distribution

As shown in section A Simple Example, you use the `sdist` command to create a source distribution. In the simplest case,

```bash
python setup.py sdist
```

(assuming you haven’t specified any `sdist` options in the setup script or config file), `sdist` creates the archive of the default format for the current platform. The default format is a gzip’ed tar file (.tar.gz) on Unix, and ZIP file on Windows.

You can specify as many formats as you like using the `--formats` option, for example:

```bash
python setup.py sdist --formats=gztar,zip
```

to create a gzipped tarball and a zip file. The available formats are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>gztar</td>
<td>gzip’ed tar file (.tar.gz)</td>
<td>(2)</td>
</tr>
<tr>
<td>bztar</td>
<td>bzip2’ed tar file (.tar.bz2)</td>
<td></td>
</tr>
<tr>
<td>xztar</td>
<td>xz’ed tar file (.tar.xz)</td>
<td></td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.Z)</td>
<td>(4)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td></td>
</tr>
</tbody>
</table>

*Changed in version 3.5:* Added support for the `xztar` format.
Notes:

1. default on Windows
2. default on Unix
3. requires either external zip utility or zipfile module (part of the standard Python library since Python 1.6)
4. requires the compress program. Notice that this format is now pending for deprecation and will be removed in the future versions of Python.

When using any tar format (gztar, bztar, xztar, ztar or tar), under Unix you can specify the owner and group names that will be set for each member of the archive.

For example, if you want all files of the archive to be owned by root:

```
python setup.py sdist --owner=root --group=root
```
4.1. Specifying the files to distribute

If you don’t supply an explicit list of files (or instructions on how to generate one), the **sdist** command puts a minimal default set into the source distribution:

- all Python source files implied by the `py_modules` and `packages` options
- all C source files mentioned in the `ext_modules` or `libraries` options
- scripts identified by the `scripts` option. See Installing Scripts.
- anything that looks like a test script: `test/test*.py` (currently, the Distutils don’t do anything with test scripts except include them in source distributions, but in the future there will be a standard for testing Python module distributions)
- Any of the standard README files (README, README.txt, or README.rst), `setup.py` (or whatever you called your setup script), and `setup.cfg`.
- all files that matches the `package_data` metadata. See Installing Package Data.
- all files that matches the `data_files` metadata. See Installing Additional Files.

Sometimes this is enough, but usually you will want to specify additional files to distribute. The typical way to do this is to write a manifest template, called `MANIFEST.in` by default. The manifest template is just a list of instructions for how to generate your manifest file, `MANIFEST`, which is the exact list of files to include in your source distribution. The **sdist** command processes this template and generates a manifest based on its instructions and what it finds in the filesystem.
If you prefer to roll your own manifest file, the format is simple: one filename per line, regular files (or symlinks to them) only. If you do supply your own MANIFEST, you must specify everything: the default set of files described above does not apply in this case.

*Changed in version 3.1:* An existing generated MANIFEST will be regenerated without sdist comparing its modification time to the one of MANIFEST.in or setup.py.

*Changed in version 3.1.3:* MANIFEST files start with a comment indicating they are generated. Files without this comment are not overwritten or removed.

*Changed in version 3.2.2:* sdist will read a MANIFEST file if no MANIFEST.in exists, like it used to do.

*Changed in version 3.7:* README.rst is now included in the list of distutils standard READMEs.

The manifest template has one command per line, where each command specifies a set of files to include or exclude from the source distribution. For an example, again we turn to the Distutils’ own manifest template:

```
include *.txt
recursive-include examples *.txt *.py
prune examples/sample?/build
```

The meanings should be fairly clear: include all files in the distribution root matching *.txt, all files anywhere under the examples directory matching *.txt or *.py, and exclude all directories matching examples/sample?/build. All of this is done *after* the standard include set, so you can exclude files from the standard set
with explicit instructions in the manifest template. (Or, you can use the
--no-defaults option to disable the standard set entirely.) There
are several other commands available in the manifest template mini-
language; see section Creating a source distribution: the sdist
command.

The order of commands in the manifest template matters: initially, we
have the list of default files as described above, and each command in
the template adds to or removes from that list of files. Once we have
fully processed the manifest template, we remove files that should not
be included in the source distribution:

- all files in the Distutils “build” tree (default build/)
- all files in directories named RCS, CVS, .svn, .hg, .git, .bzr
  or _darcs

Now we have our complete list of files, which is written to the manifest
for future reference, and then used to build the source distribution
archive(s).

You can disable the default set of included files with the --no-
defaults option, and you can disable the standard exclude set with
--no-prune.

Following the Distutils’ own manifest template, let’s trace how the
sdist command builds the list of files to include in the Distutils source
distribution:

1. include all Python source files in the distutils and
distutils/command subdirectories (because packages
  corresponding to those two directories were mentioned in the
  packages option in the setup script—see section Writing the
  Setup Script)
2. include **README.txt**, **setup.py**, and **setup.cfg** (standard files)

3. include **test/test*.py** (standard files)

4. include ***.txt** in the distribution root (this will find **README.txt** a second time, but such redundancies are weeded out later)

5. include anything matching ***.txt** or ***.py** in the sub-tree under **examples**, 

6. exclude all files in the sub-trees starting at directories matching **examples/sample*/build**—this may exclude files included by the previous two steps, so it's important that the **prune** command in the manifest template comes after the **recursive-include** command

7. exclude the entire **build** tree, and any RCS, CVS, .svn, .hg, .git, .bzr and _darcs directories

Just like in the setup script, file and directory names in the manifest template should always be slash-separated; the Distutils will take care of converting them to the standard representation on your platform. That way, the manifest template is portable across operating systems.
4.2. Manifest-related options

The normal course of operations for the `sdist` command is as follows:

- if the manifest file (`MANIFEST` by default) exists and the first line does not have a comment indicating it is generated from `MANIFEST.in`, then it is used as is, unaltered
- if the manifest file doesn't exist or has been previously automatically generated, read `MANIFEST.in` and create the manifest
- if neither `MANIFEST` nor `MANIFEST.in` exist, create a manifest with just the default file set
- use the list of files now in `MANIFEST` (either just generated or read in) to create the source distribution archive(s)

There are a couple of options that modify this behaviour. First, use the `--no-defaults` and `--no-prune` to disable the standard “include” and “exclude” sets.

Second, you might just want to (re)generate the manifest, but not create a source distribution:

```
python setup.py sdist --manifest-only
```

- `-o` is a shortcut for `--manifest-only`. 
Index – Symbols

! (exclamation)
  in a command interpreter
  in curses module
  in formatted string literal
  in glob-style wildcards, [1]
  in string formatting
  in struct format strings

! (pdb command)

!=
  operator, [1]

" (double quote)
  string literal

'''
  string literal

# (hash)
  comment, [1], [2]
  in doctests
  in printf-style formatting, [1]
  in regular expressions
  in string formatting
  source encoding declaration

$ (dollar)
  environment variables expansion
  in regular expressions
  in template strings
  interpolation in configuration files

% (percent)

-O
  command line option

-o <output>, --output= <output>
  zipapp command line option

-o, --output=<file>
  pickletools command line option

-OO
  command line option

-p <interpreter>, --python= <interpreter>
  zipapp command line option

-p, --pattern pattern
  unittest-discover command line option

-p, --preamble=<preamble>
  pickletools command line option

-p, --process
  timeit command line option

-q
  command line option
  compileall command line option

-R
  command line option

-r

`%=` augmented assignment

`& (ampersand)`
operator, [1]

`&=` augmented assignment

`' (single quote)`
string literal

```
```
string literal

`() (parentheses)`
call class definition function definition generator expression in assignment target list in printf-style formatting, [1] in regular expressions tuple display

`(`
in regular expressions

`?!`
in regular expressions

`(?#`
in regular expressions

`?:`

compileall command line option

`-r N, --repeat=N`
timeit command line option

`-R, --no-report`
trace command line option

`-r, --report`
trace command line option

`-S`
command line option

`-s`
command line option

`-s S, --setup=S`
timeit command line option

`-s, --start-directory directory`
unittest-discover command line option

`-s, --summary`
trace command line option

`-t <tarfile>`
tarfile command line option

`-t < zipfile>`
zipfile command line option

`-t, --top-level-directory directory`
unittest-discover command line option
in regular expressions
(?<=
 in regular expressions
(?)=
 in regular expressions
(?)P<
 in regular expressions
(?)P=
 in regular expressions
* (asterisk)
  function definition
  import statement
  in AST grammar
  in argparse module
  in assignment target list
  in expression lists
  in function calls, [1]
  in glob-style wildcards, [1]
  in printf-style formatting,
  [1]
  in regular expressions
  operator, [1]
**
  function definition
  in dictionary displays
  in function calls, [1]
  in glob-style wildcards
  operator, [1]
**=
  augmented assignment
*=
  augmented assignment
-t, --trace
  trace command line option
-T, --trackcalls
  trace command line option
-u
  command line option
-u, --unit=U
  timeit command line option
-V
  command line option
-v
  command line option
-v, --verbose
  tarfile command line option
  timeit command line option
  unittest-discover
  command line option
-W arg
  command line option
-X
  command line option
-x
  command line option
-x regex
  compileall command line option
. (dot)
  attribute reference
  in Tkinter
in regular expressions
+ (plus)
  binary operator, [1]
  in argparse module
  in doctests
  in printf-style formatting,
  [1]
  in regular expressions
  in string formatting
  unary operator, [1]
+=
  augmented assignment
+?
  in regular expressions
, (comma)
  argument list
  expression list, [1], [2], [3],
  [4]
  identifier list, [1]
  import statement
  in dictionary displays
  in string formatting
  in target list
  parameter list
  slicing
  tuple display
  with statement
- (minus)
  binary operator, [1]
  in doctests
  in glob-style wildcards, [1]
  in printf-style formatting,
  [1]
in regular expressions
in string formatting
unary operator, [1]
--check-hash-based-pycs
default|always|never
command line option
--create <tarfile> <source1> ...
<sourceN>
tarfile command line option
--create <zipfile> <source1> ...
<sourceN>
zipfile command line option
--details
inspect command line option
--extract <tarfile> [<output_dir]>
tarfile command line option
--extract <zipfile> <output_dir> zipfile command line option
--help
command line option
trace command line option
--ignore-dir=<dir>
trace command line option
--ignore-module=<mod>
trace command line option
--info
zipapp command line option
--invalidation-mode
[timestamp|checked-
2to3
: (colon)
annotated variable
compound statement, [1], [2], [3], [4], [5], [6]
function annotations, [1]
in SQL statements
dictionary expressions
in formatted string literal
in string formatting
lambda expression
path separator (POSIX)
slicing
; (semicolon), [1]
< (less)
in string formatting
in struct format strings
operator, [1]
<<
operator, [1]
<<=
augmented assignment
<=
operator, [1]
<BLANKLINE>
= (equals)
assignment statement
class definition
function definition
in function calls
in string formatting
in struct format strings
==
hash|unchecked-hash|
   compileall command line option
--list <tarfile>
   tarfile command line option
--list <zipfile>
   zipfile command line option
--locals
   unittest command line option
--sort-keys
   json.tool command line option
--test <tarfile>
   tarfile command line option
--test <zipfile>
   zipfile command line option
--user-base
   site command line option
--user-site
   site command line option
--version
   command line option
   trace command line option
-=
   augmented assignment
->
   function annotations, [1]
-?
   command line option
-a, --annotate
   pickletools command line option
operator, [1]
> (greater)
in string formatting
in struct format strings
operator, [1]
>=
   operator, [1]
>>
   operator, [1]
>>>= augmented assignment
>>> interpreter prompt, [1]
? (question mark)
in AST grammar
in SQL statements
in a command interpreter
in argparse module
in glob-style wildcards, [1]
in regular expressions
in struct format strings, [1]
replacement character
??
in regular expressions
@ (at)
   class definition
   function definition
 >> (greater than or equal to)
   operator
[ ] (square brackets)
in assignment target list
option
-B
  command line option
-b
  command line option
compileall command line option
-b, --buffer
  unittest command line option
-c <command>
  command line option
-c <tarfile> <source1> ... <sourceN>
  tarfile command line option
-c <zipfile> <source1> ... <sourceN>
  zipfile command line option
-c, --catch
  unittest command line option
-c, --compress
  zipapp command line option
-c, --count
  trace command line option
-C, --coverdir=<dir>
  trace command line option
-d
  command line option
-d destdir
  compileall command line option

in glob-style wildcards, [1]
in regular expressions
in string formatting
list expression
subscription
\ (backslash)
  escape sequence, [1]
in pathnames
(Windows)
in regular expressions, [1], [2]
\\
  escape sequence
\A
  in regular expressions
\a
  escape sequence
\b
  in regular expressions
\B
  escape sequence
\D
  in regular expressions
\d
  escape sequence
\f
  escape sequence
\g
-E
  command line option

-e <tarfile> [<output_dir>]
  tarfile command line option

-e <zipfile> <output_dir>
  zipfile command line option

-e, --exact
  tokenize command line option

-f
  compileall command line option

-f, --failfast
  unittest command line option

-f, --file=<file>
  trace command line option

-g, --timing
  trace command line option

-h
  command line option

-h, --help
  json.tool command line option
  timeit command line option
  tokenize command line option
  zipapp command line option

-I
  command line option

-i
  command line option

\n
escape sequence, [1]
in regular expressions

\n
escape sequence
in regular expressions

\r
escape sequence
in regular expressions

\s
escape sequence
in regular expressions

\S
escape sequence
in regular expressions

\t
escape sequence
in regular expressions

\U
escape sequence, [1]
in regular expressions

\u
escape sequence, [1]
in regular expressions

\v
escape sequence
in regular expressions

\W
escape sequence
in regular expressions

\w
escape sequence
in regular expressions

\x
escape sequence, [1]
in regular expressions

\Z
-i list
    compileall command line option
-J
    command line option
-j N
    compileall command line option
-k
    unittest command line option
-l
    compileall command line option
-l <tarfile>
    tarfile command line option
-l < zipfile>
    zipfile command line option
-l, --indentlevel=<num>
    pickletools command line option
-l, --listfuncs
    trace command line option
-m <mainfn>, --main=<mainfn>
    zipapp command line option
-m <module-name>
    command line option
-m, --memo
    pickletools command line option
-m, --missing
    trace command line option

in regular expressions
^ (caret)
in curses module
in regular expressions,
[1]
in string formatting
marker, [1]
operator, [1]

^=
    augmented assignment
{}
    dictionary expression

{} (curly brackets)
dictionary expression
in formatted string literal
in regular expressions
in string formatting
set expression

| (vertical bar)
in regular expressions
operator, [1]

|=    augmented assignment

~ (tilde)
home directory
expansion
operator, [1]
-n N, --number=N

timeit command line option
Index – __

__(underscore) (natural attribute)  

gettext
  in numeric literal, [1]
  in string formatting

__, identifiers
  __identifiers (natural attribute)

__abs__() (in module operator)
  (object method)

__add__() (in module operator)
  (object method)

__aenter__() (object method)

__aexit__() (object method)

__aiter__() (object method)

__all__
  (optional module attribute)
  (package variable)

__and__() (in module operator)
  (object method)

__anext__() (agen method)
  (object method)

__annotations__ (class attribute)
  (function attribute)
  (module attribute)

__await__() (object method)

__bases__ (class attribute), [1]

__bool__() (object method), [1]

__breakpointhook__ (in module sys)

__bytes__()
  (email.message.EmailMessage method)
  (email.message.Message

__name__
  (class attribute)

__ne__() (email.charset.Charset method)

__new__() (object method)

__next__() (csv.csvreader method)
  (generator method)

__not__() (in module operator)

__or__() (in module operator)
  (object method)

__package__
  (module attribute)
  (types.ModuleType attribute)

__path__

__pos__() (in module operator)
  (object method)

__pow__() (in module operator)
  (object method)

__prepare__ (metaclass)

__qualname__ (definition attribute)
method)
    (object method)
    __cached__
__call__()
(email.headerregistry.HeaderRegistry method)
    (object method), [1]
    (weakref.finalize method)
    __callback__ (weakref.ref attribute)
    __cause__ (exception attribute)
        (traceback.TracebackException attribute)
    __ceil__() (fractions.Fraction method)
    (object method)
    __class__ (instance attribute), [1]
        (method cell)
        (module attribute)
        (unittest.mock.Mock attribute)
    __class_getitem__() (object class method)
    __classcell__ (class namespace entry)
    __closure__ (function attribute)
    __code__ (function attribute)
        (function object attribute)
    __complex__() (object method)
    __concat__() (in module operator)
    __contains__()
(email.message.EmailMessage method)
    (email.message.Message method)
    (in module operator)
    __radd__() (object method)
    __rand__() (object method)
    __rdivmod__() (object method)
    __reduce__() (object method)
    __reduce_ex__() (object method)
    __repr__() (multiprocessing.manager method)
    __reversed__() (object method)
    __rfloordiv__() (object method)
    __rlshift__() (object method)
    __rmatmul__() (object method)
    __rmod__() (object method)
    __rmul__() (object method)
    __ror__() (object method)
    __round__() (fractions.Fraction method)
    (object method)
    __rpow__() (object method)
    __rrshift__() (object method)
    __rsub__() (object method)
    __rtruediv__() (object method)
    __rxor__() (object method)
    __self__ (method attribute)
    __set__() (object method)
    __set_name__() (object method)
    __setattr__() (object method)
    __setitem__() (object method)
    (email.message.EmailMessage method)
    (email.message.Message method)
    (in module operator)
(mailbox.Mailbox method)
(mailbox.Mailbox method)

__context__ (exception attribute)
(traceback.TracebackException attribute)

__copy__() (copy protocol)

__debug__
(built-in variable)

__deepcopy__() (copy protocol)

__defaults__ (function attribute)

__del__() (io.IOBase method)
(object method)

__delattr__() (object method)

__delete__() (object method)

__delitem__()
(email.message.EmailMessage method)

(email.message.Message method)
(mailbox.MH method)
(mailbox.Mailbox method)

__dict__ (class attribute)

__dir__ (module attribute)

__dir__() (object method)

__displayhook__ (in module sys)

__divmod__() (object method)

__doc__ (class attribute)

__setstate__() (copy protocol)

__slots__

__spec__

__stderr__ (in module sys)

__stdin__ (in module sys)

__stdout__ (in module sys)

__str__() (datetime.datetime method)

(email.charset.Charset method)

(email.header.Header method)

(email.headerregistry.Address method)

(email.headerregistry.Group method)

(email.message.EmailMessage method)

(email.message.Message method)

(in module operator)

(mailbox.Maildir method)

(mailbox.Mailbox method)

__subclasscheck__() (class method)

__subclasses__() (class method)

__subclasshook__() (class method)

__suppress_context__
(traceback.TracebackException attribute)

__traceback__ (exception attribute)

__truediv__() (in module operator)
__future__
    future statement
__future__ (module)
__ge__() (in module operator)
    (instance method)
    (object method)
__get__() (object method)
__getattr__() (module attribute)
__getattr__() (object method)
__getattribute__() (object method)
__getitem__()
(email.headerregistry.HeaderRegistry method)
    (email.message.EmailMessage method)
    (email.message.Message method)
    (in module operator)
    (mailbox.Mailbox method)
    (mapping object method)
    (object method)
    (re.Match method)
__getnewargs__() (object method)
__getnewargs_ex__() (object method)
__getstate__() (copy protocol)
    (object method)
__globals__ (function attribute)
__gt__() (in module operator)
    (instance method)
    (object method)
__hash__() (object method)
__iadd__() (in module operator)
    (object method)
(multiprocessing.managers.BaseProxy method)
__handle (ctypes.PyDLL method)
__inittab (C type)
__length_ (ctypes.Array attribute)
_locale
    module
__make__() (collection class method)
__makeResult__() (unittest method)
__name (ctypes.PyDLL attribute)
__objects (ctypes._CData pointer)
__pack__ (ctypes.Structure method)
__parse__() (gettext.NullTranslations method)
__Pointer (class in ctypes)
__Py_c_diff (C function)
__Py_c_neg (C function)
__Py_c_pow (C function)
__Py_c_prod (C function)
__Py_c_quot (C function)
__Py_c_sum (C function)
__Py_NoneStruct (C variable)
__PyBytes_Resize (C function)
__PyImport_Fini (C function)
__PyImport_Init (C function)
__PyObject_GC_TRACK (C function)
__PyObject_GC_UNTRACK (C function)
__PyObject_New (C function)
__PyObject_NewVar (C function)
__PyTuple_Resize (C function)
_replace() (collections.somenamedtuple method)
_setroot()
__ipow__() (in module operator)  
  (object method)
__irshift__() (in module operator)  
  (object method)
__isub__() (in module operator)  
  (object method)
__iter__() (container method)  
  (iterator method)
  (mailbox.Mailbox method)  
  (object method)
  (unittest.TestSuite method)
__itruediv__() (in module operator)  
  (object method)
__ixor__() (in module operator)  
  (object method)
__kwdefaults__ (function attribute)  
__le__() (in module operator)  
  (instance method)
  (object method)
__len__()  
  (email.message.EmailMessage method)  
    (email.message.Message method)  
    (mailbox.Mailbox method)  
    (mapping object method)  
    (object method)
__length_hint__() (object method)  
__loader__  
  (module attribute)  
    (types.ModuleType attribute)
__lshift__() (in module operator)  
  (object method)
__lt__() (in module operator)
(instance method)
(object method)
__main__
    module, [1], [2], [3], [4], [5], [6], [7]
__main__ (module)
__matmul__() (in module operator)
    (object method)
__missing__()  
    (collections.defaultdict method)
    (object method)
__mod__() (in module operator)
    (object method)
__module__ (class attribute)
    (function attribute)
    (method attribute)
__mro__ (class attribute)
__mul__() (in module operator)
    (object method)
Index – A

A (in module re)
A-LAW, [1]
a-LAW
a2b_base64() (in module binascii)
a2b_hex() (in module binascii)
a2b_hqx() (in module binascii)
a2b_qp() (in module binascii)
a2b_uu() (in module binascii)
a85decode() (in module base64)
a85encode() (in module base64)
ABC (class in abc)
abc (module)
ABCMeta (class in abc)
abiflags (in module sys)
abort()
  (asyncio.DatagramTransport method)
  (asyncio.WriteTransport method)
  (ftplib.FTP method)
  (in module os)
  (threading.Barrier method)
above() (curses.panel.Panel method)
ABOVE_NORMAL_PRIORITY_CLASS (in module subprocess)
abs
  built-in function, [1]
abs() (built-in function)
  (decimal.Context method)
  (in module operator)
abspath() (in module os.path)
abstract base class
AbstractAsyncContextManager (class in contextlib)
(threading.Semaphore method) acquire_lock() (in module imp)

Action (class in argparse) action (optparse.Option attribute)

ACTIONS (optparse.Option attribute) ACTION	(class in argparse)

active_children() (in module multiprocessing) add_argument()

active_count() (in module threading) add_argument_group()

add() (decimal.Context method) add_attachment()

(frozenset method) add_cgi_vars() (wsgiref.handlers.BaseHandler

(in module audioop) add_charset() (in module email.charset)

(in module operator) add_child_handler() (asyncio.AbstractChildWatcher

(mailbox.Mailbox method) add_codec() (in module email.charset)

(mailbox.Maildir method) add_cookie_header() (http.cookiejar.CookieJar

(msilib.RadioButtonGroup method) add_data() (in module msilib)

(pstats.Stats method) ASCII,

(tarfile.TarFile method) ascii, [1]

(tkinter.ttk.Notebook method) ascii,

(add_alias() (in module email.charset)

(add_alternative() (email.message.EmailMessage

method) add_cgi_vars() (wsgiref.handlers.BaseHandler

method) add_child_handler() (asyncio.AbstractChildWatcher

method) add_cookie_header() (http.cookiejar.CookieJar

method) add_data() (in module msilib)

(AS_IS (in module formatter)

(ascii() (in module re)

(ascii_letters (in module string)

(ascii_lowercase (in module string)

(ascii_uppercase (in module string)

(asctime() (in module time)

(asdict() (in module dataclasses)

(asend() (agen method)

(asin() (in module cmath)

(asinh() (in module cmath)

(assert) statement

(with statement)
add_done_callback() (asyncio.Future method) assert_a
    (asyncio.Task method) method)
    (concurrent.futures.Future method) assert_c
add_fallback() (gettext.NullTranslations method) assert_c
add_file() (msilib.Directory method) method)
add_flag() (mailbox.MaildirMessage method) assert_c
    (mailbox.MMDFMessage method) (unittest.
    (mailbox.mboxMessage method) method)
add_flowing_data() (formatter.formatter method) method)
add_folder() (mailbox.Maildir method) assert_h
    (mailbox.MH method) method)
add_get_handler() assert_l
    (email.contentmanager.ContentManager method) method)
add_handler() (urllib.request.OpenerDirector method)
add_header() (email.message.EmailMessage assert_p
    (email.message.Message method) test.sup
    (urllib.request.Request method) test.sup
    (wsgiref.headers.Headers method) assertAli
add_history() (in module readline) method)
add_hor_rule() (formatter.formatter method) assertC
add_include_dir() (distutils.ccompiler.CCompiler method)
add_label() (mailbox.BabylMessage method) assertEc
add_label_data() (formatter.formatter method) assertFa
add_library() (distutils.ccompiler.CCompiler method) assertGr
add_library_dir() (distutils.ccompiler.CCompiler method)
add_line_break() (formatter.formatter method) assertIn(
add_link_object() (distutils.ccompiler.CCompiler Assertions
    (distutils.ccompiler.CCompiler method)
add_literal_data() (formatter.formatter method) exception
add_mutually_exclusive_group()
addUnexpectedSuccess()	 (unittest.TestResult method)
adjusted() (decimal.Decimal method)
adler32() (in module zlib)
ADPCM, Intel/DVI
adpcm2lin() (in module audioop)
AF_ALG (in module socket)
AF_CAN (in module socket)
AF_INET (in module socket)
AF_INET6 (in module socket)
AF_LINK (in module socket)
AF_PACKET (in module socket)
AF_RDS (in module socket)
AF_UNIX (in module socket)
AF_VSOCK (in module socket)
aifc (module)
aifc() (aifc.aifc method)
AIFF, [1]
aiff() (aifc.aifc method)
AIIFF-C, [1]
alarm() (in module signal)
alaw2lin() (in module audioop)
ALER_DESCRIPTION_HANDSHAKE_FAILURE (in module ssl)
ALER_DESCRIPTION_INTERNAL_ERROR (in module ssl)
AlertDescription (class in ssl)
algorithms_available (in module hashlib)
algorithms_guaranteed (in module hashlib)
alias (pdb command)
alignment() (in module ctypes)
alive (weakref.finalize attribute)
all() (built-in function)
all_errors (in module ftplib)
all_features (in module xml.sax.handler)
all_frames (tracemalloc.Filter attribute)
all_properties (in module xml.sax.handler)
all_suffixes() (in module importlib.machinery)
all_tasks() (asyncio.Task class method)
  (in module asyncio)
allocate_lock() (in module _thread)
allow_reuse_address (socketserver.BaseServer attribute)
allowed_domains()
  (http.cookiejar.DefaultCookiePolicy method)
alt() (in module curses.ascii)
ALT_DIGITS (in module locale)
altsep (in module os)
altzone (in module time)
ALWAYS_TYPED_ACTIONS (optparse.Option attribute)
AMPER (in module token)
AMPEREQUAL (in module token)
and
  bitwise
  operator, [1], [2]
and_() (in module operator)
annotated
  annotation
  assignment
annotation
  (inspect.Parameter attribute)
annotations
  function, [1]
announce() (distutils.ccompiler.CCompiler method)
anonymous
  function
answer_challenge() (in module multiprocessing.connection)
athrow() (agen method)
atof() (in module locale)
atoi() (in module locale)
atom
attach()
  (in module asyncio)
attach_lock()
allow_reuse_address
attrib
ALT_DIGITS
attrib
and
bitwise
operator, [1], [2]
and_()
annotated
annotation
annotations
function, [1]
announce()
anonymous
function
answer_challenge()
anticipate_failure() (in module test.support)
Any (in module typing)
ANY (in module unittest.mock)
any() (built-in function)
AnyStr (in module typing)
api_version (in module sys)

AUDIO_FILE_ENCODING_ADPCM_G722 (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G723_3 (in module sunau)
AUDIO_FILE_ENCODING_ADPCM_G723_5 (in module sunau)
AUDIO_FILE_ENCODING_ALAW_8 (in module sunau)
AUDIO_FILE_ENCODING_DOUBLE (in module sunau)
AUDIO_FILE_ENCODING_FLOAT (in module sunau)
AUDIO_FILE_ENCODING_LINEAR_16 (in module sunau)
AUDIO_FILE_ENCODING_LINEAR_24 (in module sunau)
AUDIO_FILE_ENCODING_LINEAR_32 (in module sunau)
AUDIO_FILE_ENCODING_LINEAR_8 (in module sunau)
AUDIO_FILE_ENCODING_MULAW_8 (in module sunau)
AUDIO_FILE_MAGIC (in module sunau)
AUDIODEV
audioop
augmented
auth() (ftplib.FTP_TLS method)
(auth() (smtplib.SMTP method)
authenticate() (imaplib.IMAP4 method)
AuthenticationError
authenticators() (netrc.netrc method)
authkey (multiprocessing.Process attribute)
auto (class in enum)
| b" |
| b' |
| b16decode() |
| b16encode() |
| b2a_base64() |
| b2a_hex() |
| b2a_hqx() |
| b2a_qp() |
| b2a_uu() |
| b32decode() |
| b32encode() |
| b64decode() |
| b64encode() |
| b85decode() |
| b85encode() |
| Babyl |
| BabylMessage |
| back() |
| backslash replace_errors() |
| backup() |
| backward() |
| BadStatusLine |
| BadZipFile |
| BadZipfile |
| Balloon |
| Barrier |
| blake2s.SALT_SIZE |
| blank line |
| block |
| code |
| block_size |
| blocked_domains() |
| BlockingIOError |
| body() |
| body_encoding |
| body_line_iterator() |
| BOM |
| BOM_BE |
| BOM_LE |
| BOM_UTF16 |
| BOM_UTF16_BE |
| BOM_UTF16_LE |
| BOM_UTF32 |
| BOM_UTF32_BE |
| BOM_UTF32_LE |
| BOM_UTF8 |
| bool |
| Boolean |
Barrier()
(multiprocessing.managers.SyncManager method)
base64
coding
module
base64 (module)
base_exec_prefix (in module sys)
base_prefix (in module sys)
BaseCGIHandler (class in wsgiref.handlers)
BaseCookie (class in http.cookies)
BaseException
BaseHandler (class in urllib.request)
BaseHeader (class in wsgiref.handlers)
BaseHTTPRequestHandler (class in http.server)
BaseManager (class in multiprocessing.managers)
basename() (in module os.path)
BaseProtocol (class in asyncio)
BaseProxy (class in multiprocessing.managers)
BaseRequestHandler (class in socketserver)
BaseRotatingHandler (class in logging.handlers)
BaseSelector (class in selectors)
BaseServer (class in socketserver)
basestring (2to3 fixer)
BaseTransport (class in asyncio)
basicConfig() (in module logging)
BasicContext (class in decimal)
BasicInterpolation (class in configparser)
BasicTestRunner (class in test.support)
baudrate() (in module curses)
bbox() (tkinter.ttk.Treeview method)
BDADDR_ANY (in module socket)
BDADDR_LOCAL (in module socket)
bdb
    module
Bdb (class in bdb)
bdb (module)
BdbQuit
BDFL
bdist_msi (class in distutils.command.bdist_msi)
beep() (in module curses)
Beep() (in module winsound)
BEFORE_ASYNC_WITH (opcode)
begin_fill() (in module turtle)
begin_poly() (in module turtle)
bellow() (curses.panel.Panel method)
BELOW_NORMAL_PRIORITY_CLASS (in module subprocess)
Benchmarking
benchmarking, [1], [2], [3]
betavariate() (in module random)
bgcolor() (in module turtle)
bgpic() (in module turtle)
bias() (in module audioop)
bidirectional() (in module unicodedata)
bigaddrspacetest() (in module test.support)
BigEndianStructure (class in ctypes)
bigmemtest() (in module test.support)
breakpointhook() (in breakpoints)
broadcast_address (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Network attribute)
broken (threading.Executor)
BrokenBarrierError
BrokenExecutor
BrokenPipeError
BrokenProcessPool
BrokenThreadPool
BROWSER, [1]
BsdDbShelf (class in shelve)
buffer (2to3 fixer)
    (io.TextIOBase attribute)
    (unittest.TestResult attribute)
buffer interface
    (see buffer protocol)
buffer object
    (see buffer protocol)
buffer protocol
    binary protocol
    buffer size, I/O
    buffer_info() (array.array attribute)
    buffer_size (xml.parsers.expat.xmlparser attribute)
    buffer_text (xml.parsers.expat.xmlparser attribute)
    buffer_updated() (asyncio.BufferedProtocol method)
    buffer_used (xml.parsers.expat.xmlparser attribute)
BufferedIOBase (class in io)
bin() (built-in function)
binary
  arithmetic operation
  bitwise operation
  data, packing
  literals
Binary (class in msilib)
  (class in xmlrpc.client)
binary file
binary literal
binary mode
binary semaphores
BINARY_ADD (opcode)
BINARY_AND (opcode)
BINARY_FLOOR_DIVIDE (opcode)
BINARY_LSHIFT (opcode)
BINARY_MATRIX_MULTIPLY (opcode)
BINARY_MODULO (opcode)
BINARY_MULTIPLY (opcode)
BINARY_OR (opcode)
BINARY_POWER (opcode)
BINARY_RSHIFT (opcode)
BINARY_SUBSCR (opcode)
BINARY_SUBTRACT (opcode)
BINARY_TRUE_DIVIDE (opcode)
BINARY_XOR (opcode)
BinaryIO (class in typing)
binascii (module)
bind (widgets)
bind() (asyncore.dispatcher method)
  (inspect.Signature method)
  (socket.socket method)
bind_partial() (inspect.Signature method)
bind_port() (in module test.support)
BufferedProtocol (c
BufferedRandom (c
BufferedReader (cl
BufferedRWPair (cl
BufferedWriter (cl
BufferError
BufferingHandler
logging.handlers
BufferTooShort
bufsize() (ossaudio method)
BUILD_CONST_KEY_MAP
BUILD_LIST
BUILD_LIST_UNPACK
BUILD_MAP
BUILD_MAP_UNPACK
BUILD_MAP_UNPACK_WITH_CALL
BUILD_SET
BUILD_SET_UNPACK
BUILD_SLICE
BUILD_STRING
BUILD_TUPLE
BUILD_TUPLE_UNPACK
BUILD_TUPLE_UNPACK_WITH_CALL
built-in
built-in
bind_textdomain_codeset() (in module gettext)
bind_unix_socket() (in module test.support)
binding
  global name
    name, [1], [2], [3], [4], [5], [6]
bindtextdomain() (in module gettext)
  (in module locale)
binhex
  module
binhex (module)
binhex() (in module binhex)
bisect (module)
bisect() (in module bisect)
bisect_left() (in module bisect)
bisect_right() (in module bisect)
bit_length() (int method)
bitmap() (msilib.Dialog method)
bitwise
  and
    operation, binary
    operation, unary
    operations
  or
    xor
bk() (in module turtle)
bkgd() (curses.window method)
bkgdset() (curses.window method)
blake2b() (in module hashlib)
blake2b, blake2s
blake2b.MAX_DIGEST_SIZE (in module hashlib)
blake2b.MAX_KEY_SIZE (in module hashlib)
built-in function
  __import__
    abs, [1]
    ascii
    bytes, [1]
    call
    chr
    classmethod
    compile, [1], [2]
    complex, [1]
    divmod, [1], [2]
    eval, [1], [2], [3]
    exec, [1], [2], [:]
    float, [1], [2]
    hash, [1], [2], [:]
    help
    id
    int, [1], [2]
    len, [1], [2], [3]
    [9], [10], [11]
    max
    min
    object, [1]
    open, [1]
    ord
    pow, [1], [2], [3]
    print
    range
    repr, [1], [2], [3]
    round
    slice, [1]
    staticmethod
    tuple, [1]
    type, [1], [2], [3]
hashlib)
blake2b.PERSON_SIZE (in module hashlib)
blake2b.SALT_SIZE (in module hashlib)
blake2s() (in module hashlib)
blake2s.MAX_DIGEST_SIZE (in module hashlib)
blake2s.MAX_KEY_SIZE (in module hashlib)
blake2s.PERSON_SIZE (in module hashlib)

built-in method
call
object, [1]
builtin_module_names (in module sys)
BuiltinFunctionType
BuiltinImporter
importlib.machinery
BuiltinMethodType
builtins
module, [1], [2]
builtins (module)
ButtonBox (class in tkinter.tix)
bye() (in module turtle)
byref() (in module ctypes)
byte
byte-code
file, [1]
byte_compile() (in module code)
bytearray
formatting
interpolation
methods
object, [1], [2],
bytearray (built-in class)
bytecode, [1]
Bytecode (class in dis)
Bytecode.codeobj (in module dis)
Bytecode.first_line (in module dis)
BYTECODE_SUFFIXES (in module importlib.machinery)
byteorder (in module bytes)
built-in function
formatting
interpolation methods

object, [1], [2]
str (built-in class)
bytes (built-in class)
    (uuid.UUID attribute)
bytes literal

**bytes-like object**

bytes_le (uuid.UUID attribute)
BytesFeedParser (class in email.parser)
BytesGenerator (class in email.generator)
BytesHeaderParser (class in email.parser)
BytesIO (class in io)
BytesParser (class in email.parser)
ByteString (class in typing)
    (class in typing)
byteswap() (array.array method)
    (in module audioop)

BytesWarning
bz2 (module)
BZ2Compressor (class in bz2)
BZ2Decompressor (class in bz2)
BZ2File (class in bz2)
Index – C

C

- language, [1], [2], [3], [4], [5]
- structures

C-contiguous, [1]
c_bool (class in ctypes)
C_BUILTIN (in module imp)
c_byte (class in ctypes)
c_char (class in ctypes)
c_char_p (class in ctypes)
c_contiguous (memoryview attribute)
c_double (class in ctypes)
C_EXTENSION (in module imp)
c_float (class in ctypes)
c_int (class in ctypes)
c_int16 (class in ctypes)
c_int32 (class in ctypes)
c_int64 (class in ctypes)
c_int8 (class in ctypes)
c_long (class in ctypes)
c_longdouble (class in ctypes)
c_longlong (class in ctypes)
c_short (class in ctypes)
c_size_t (class in ctypes)
c_ssize_t (class in ctypes)
c_ubyte (class in ctypes)
c_uint (class in ctypes)
c_uint16 (class in ctypes)
c_uint32 (class in ctypes)
c_uint64 (class in ctypes)
c_uint8 (class in ctypes)
c_ulong (class in ctypes)

COLON (in module token)
color() (in module turtle)
color_content() (in module curses)
color_pair() (in module curses)
colormode() (in module curses)
COLORS, [1]
column() (tkinter.ttk method)
COLUMNS, [1]
columns (os.terminal_size attribute)
combinations() (in module itertools)
combinations_with_replacement() (in module itertools)
combine() (datetime.datetime class method)
combining() (in module unicodedata)
ComboBox (class in tkinter.tix)
Combobox (class in tkinter.ttk)
comma
  trailing tuple display
COMMA (in module token)
Command (class in distutils.cmd)
  (class in distutils.core)
command
  command line
command line op
    --check-hash
  default|always
    --help
c_ulonglong (class in ctypes) --version
c_ushort (class in ctypes) -?
c_void_p (class in ctypes) -B
c_wchar (class in ctypes) -E
c_wchar_p (class in ctypes) -I
CAB (class in msilib) -J

CacheFTPHandler (class in urllib.request) -V
calcobjsize() (in module test.support) -W arg
calcsiz() (in module struct) -X
calcvobjsize() (in module test.support) -b
Calendar (class in calendar) -c <command>
calendar (module) -d
calendar() (in module calendar) -h
call
    built-in function -m <module-'
built-in method -q
class instance -s
class object, [1], [2] -u
function, [1], [2] -v
instance, [1] -x
method CommandCompiler

procedure commands (pdb commands (pdb
user-defined function comment

call() (in module subprocess) (http.cookiejar
    (in module unittest.mock)
    COMMENT (in m

call_args (unittest.mock.Mock attribute) comment (zipfile.

call_args_list (unittest.mock.Mock attribute) comment (zipfile.
call_at() (asyncio.loop method) Comment() (in m

call_count (unittest.mock.Mock attribute) comment_url (http

call_exception_handler() (asyncio.loop commenters (shlex.

...
method)
CALL_FUNCTION (opcode)
CALL_FUNCTION_EX (opcode)
CALL_FUNCTION_KW (opcode)
callLater() (asyncio.loop method)
callList() (unittest.mock.call method)
CALL_METHOD (opcode)
callSoon() (asyncio.loop method)
callSoonThreadsafe() (asyncio.loop method)
callTracing() (in module sys)
callable
    object, [1]
Callable (class in collections.abc)
    (in module typing)
callable() (built-in function)
CallableProxyType (in module weakref)
callback (optparse.Option attribute)
callback() (contextlib.ExitStack method)
callback_args (optparse.Option attribute)
callback_kwargs (optparse.Option attribute)
callbacks (in module gc)
called (unittest.mock.Mock attribute)
CalledProcessError
calloc()
CAN_BCM (in module socket)
can_change_color() (in module curses)
can_fetch()
(canlib.robotparser.RobotFileParser method)
CAN_ISOTP (in module socket)
CAN_RAW_FD_FRAMES (in module socket)
can_symlink() (in module test.support)
can_write_eof() (asyncio.StreamWriter
CommentHandler() (xml.parsers.expat.
commit() (msilib.CAB method)
Commit() (msilib.Database method)
commit() (sqlite3.Connection method)
common (filecmp)
Common Gateway Interface
common_dirs (filecmp)
common_files (filecmp)
common_funny ()
common_types (typing)
commonpath() (in module os.path)
commonprefix() (in module os.path)
communicate() (asyncio.asyncio.
subprocess)
compare() (decimal)
compare() (difflib)
cmpare_digest()
compare_network()
compare_to() (tracemalloc.Snapshot
compare_total() (decimal)
compare_total_mag() (decimal)
comparing
method) (asyncio.WriteTransport method)
can_xattr() (in module test.support)
cancel_join_thread() (multiprocessing.Queue method)
casefold() (str method)
cast() (in module ctypes)
    (in module typing)
    (memoryview method)
cat() (in module nis)
catch_warnings (class in warnings)
category() (in module unicodedata)
cbreak() (in module curses)
CC
ccc() (ftplib.FTP_TLS method)
CCompiler (class in distutils.ccompiler)
CDLL (class in ctypes)
ceil() (in module math), [1]
center() (bytearray method)
    (bytes method)
    (str method)
CERT_NONE (in module ssl)
CERT_OPTIONAL (in module ssl)
CERT_REQUIRED (in module ssl)
cert_store_stats() (ssl.SSLContext method)
cert_time_to_seconds() (in module ssl)
CertificateError
certificates
CFLAGS, [1], [2]
CFUNCTYPE() (in module ctypes)
CGI
    debugging
    exceptions
    protocol
    security
    tracebacks
cgi (module)
cgi_directories
(http.server.CGIHTTPRequestHandler
directory ...
file ...
compilest() (in module parser)
complete() (rlcompleter.Completer method)
complete_statement()
completedefault()
CompletedProcess
complex
    built-in function
    number
    object
    complex (built-in object)
    Complex (class in numbers)
    complex literal
    complex_numbe
    literals
    object, [1]
compound
    statement
    comprehensions
    list
    compress() (bz2.lzma.LZMACompressor method)
    (in module bz2)
    (in module gzip)
    (in module itertools)
    (in module lzma)
    (zlib.Compress method)
    write_size (zipfile.ZipInfo attribute)
    compress_type (zipfile.ZipInfo attribute)
    compressed (ipaddress.IPv4Address attribute)
    (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Address attribute)
attribute)
CGIHandler (class in wsgiref.handlers)
CGIHTTPRequestHandler (class in http.server)
cgitb (module)
CGIXMLRPCRequestHandler (class in xmlrpc.server)
chain() (in module itertools)
chaining
  comparisons, [1]
  exception
ChainMap (class in collections)
  (class in typing)
change_cwd() (in module test.support)
change_root() (in module distutils.util)
CHANNEL_BINDING_TYPES (in module ssl)
channel_class (smtpd.SMTPServer attribute)
channels() (ossaudiodev.oss_audio_device method)
CHAR_MAX (in module locale)
character, [1], [2]
CharacterDataHandler()
  (xml.parsers.expat.xmlparser method)
characters()
  (xml.sax.handler.ContentHandler method)
characters_written (BlockingIOError attribute)
Charset (class in email.charset)
charset() (gettext.NullTranslations method)
chdir() (in module os)
check (lzma.LZMADEcompressor attribute)
check() (imaplib.IMAP4 method)
(ipaddress.IG
compression() (ss
CompressionError
compressobj() (in
COMSPEC, [1]
concat() (in module
concatenation
  operation
concurrent.futures:
Condition (class i
  (class in multi
  (class in thre
condition (pdb co
condition() (msilib:
Condition()
  (multiprocessing.
method)
Conditional
  expression
conditional
  expression
ConfigParser (cla
configparser (mo
configuration
  file
  file, debugge
  file, path
  configuration info
configure() (tkinte
configure_mock()
confstr() (in modu
confstr_names (in
conjugate() (compr
  (decimal.Dec
children (pyclbr.Class attribute)  
   (pyclbr.Function attribute)  
chmod() (in module os)  
   (pathlib.Path method)  
choice() (in module random)  
   (in module secrets)  
choices (optparse.Option attribute)  
choices() (in module random)  
chown() (in module os)  
   (in module shutil)  
chr  
   built-in function  
chr() (built-in function)  
chroot() (in module os)  
Chunk (class in chunk)  
chunk (module)  
cipher  
   DES  
cipher() (ssl.SSLSocket method)  
circle() (in module turtle)  
CIRCUMFLEX (in module token)  
CIRCUMFLEXEQUAL (in module token)  
Clamped (class in decimal)  
class  
   attribute  
   attribute assignment  
   body  
   constructor  
   definition, [1]  
   instance  
   name  
   object, [1], [2]  
   statement  
Class (class in symtable)  
attribute  
container, [1]  
iteration over  
Container (class in typii  
contains() (in module  
content_type  
MIME  
content_manager  
attribute)  
content_type  
(email.headerregi  
attribute)  
ContentDispositi  
email.headerregis  
ContentHandler (  
ContentManager  
email.contentmar  
contents (ctypes.  
contents() (in  
method)  
(email.in  
ContentTooShortl  
ContentTransferE  
email.headerregi  
ContentTypeHe  
email.headerregi  
Context (class in  
(class in deci  
context (ssl.SSLS  
context manager  
context manager,  
context_diff() (in  
ContextDecorator
Class browser
class instance
  attribute
  attribute assignment
call
  object, [1], [2]
class object
  call, [1], [2]
**class variable**
classmethod
  built-in function
classmethod() (built-in function)
ClassMethodDescriptorType (in module types)
ClassVar (in module typing)
clause
CLD_CONTINUED (in module os)
CLD_DUMPED (in module os)
CLD_EXITED (in module os)
CLD_TRAPPED (in module os)
clean() (mailbox.Maildir method)
cleandoc() (in module inspect)
CleanImport (class in test.support)
cleanup functions
clear (pdb command)
Clear Breakpoint
clear() (asyncio.Event method)
  (collections.deque method)
  (curses.window method)
  (dict method)
  (email.message.EmailMessage method)
  (frame method)
  (frozenset method)
  contextlib (module)
  ContextManager
  contextmanager()
  ContextVar (class
  contextvars (module
  contextvars.Token (class
  contiguous, [1]
  (memoryview
continue
  statement, [1]
continue (pdb core
CONTINUE_LOC
Control (class in \n  (class in tkinter.
control() (msilib.
  (select.kqueue
controlnames (in
controls() (method)
conversion
  arithmetic
  string, [1]
ConversionError
conversions
  numeric
convert_arg_line_
  (argparse.ArgumentParser
convert_field() (string
convert_path() (in
Cookie (class in \n  CookieError
CookieJar (class
  cookiejar (urllib.re
  attribute)
clock_getres() (in module time)
clock_gettime() (in module time)
clock_gettime_ns() (in module time)
CLOCK_HIGHRES (in module time)
CLOCK_MONOTONIC (in module time)
CLOCK_MONOTONIC_RAW (in module time)
CLOCK_PROCESS_CPUTIME_ID (in module time)
CLOCK_PROF (in module time)
CLOCK_REALTIME (in module time)
clock_settime() (in module time)
clock_settime_ns() (in module time)
CLOCK_THREAD_CPUTIME_ID (in module time)
CLOCK_UPTIME (in module time)
clone() (email.generator.BytesGenerator method)
    (email.generator.Generator method)
    (email.policy.Policy method)
    (in module turtle)
    (pipes.Template method)
cloneNode() (xml.dom.Node method)
close() (aifc.aifc method), [1]
    (asyncio.AbstractChildWatcher method)
    (asyncio.BaseTransport method)
    (asyncio.Server method)
    (asyncio.StreamWriter method)
    (asyncio.SubprocessTransport method)
    (asyncio.loop method)
    (asyncore.dispatcher method)
    (chunk.Chunk method)
copy_tree() (in module distutils.dir_util)
copyfile() (in module shutil)
copyfileobj() (in module shutil)
copymode() (in module shutil)
copyreg (module)
copyright (built-in variable)
    (in module sys)
copysign() (in module math)
copystat() (in module shutil)
copystate() (in module shutil)
copytree() (in module shutil)
coroutine, [1], [2] coroutine function
    Coroutine (class in collections.abc)
    Coroutine (class in typing)
coroutine() (in module asyncio)
coroutineype (in module asyncio)
cos() (in module cmath)
cosh() (in module cmath)
count (tracemalloc.Statistic attribute)
count () (array.array method)
count() (bytearray method)
count() (collections.Counter method)
count() (collections.deque method)
count() (sequence method)
cout_diff (traceme
(multiprocessing.connection.Connection method)
   (multiprocessing.Process method)
   (multiprocessing.Queue method)
   (multiprocessing.connection.Listener method)
   (multiprocessing.pool.Pool method)
   (os.scandir method)
   (ossaudiodev.oss_audio_device method)
   (ossaudiodev.oss_mixer_device method)
   (select.devpoll method)
   (select.epoll method)
   (select.kqueue method)
   (selectors.BaseSelector method)
   (shelve.Shelf method)
   (socket.socket method)
   (sqlite3.Connection method)
   (sqlite3.Cursor method)
   (sunau.AU_read method)
   (sunau.AU_write method)
   (tarfile.TarFile method)
   (telnetlib.Telnet method)
   (urllib.request.BaseHandler method)
   (wave.Wave_read method)
   (wave.Wave_write method)
Close() (winreg.PyHKEY method)
close() (xml.etree.ElementTree.TreeBuilder method)
   (xml.etree.ElementTree.XMLParser method)
   (xml.etree.ElementTree.XMLPullParser method)
create_datagram_endpoint() (asyncio.loop method)
create_decimal() (decimal.Context method)
create_decimal_from_float() (decimal.Context method)
create_default_context() (in module ssl)
CREATE_DEFAULT_ERROR_MODE (in module subprocess)
create_empty_file() (in module test.support)
create_function() (sqlite3.Connection method)
create_future() (asyncio.loop method)
create_module (C function)
create_module() (importlib.machinery.ExtensionFileLoader method)
CREATE_NEW_CONSOLE (in module subprocess)
CREATE_NEW_PROCESS_GROUP (in module subprocess)
CREATE_NO_WINDOW (in module subprocess)
create_server() (asyncio.loop method)
create_shortcut() (create_string_buffer() (in module ctypes)
create_socket() (create_subprocess_exec() (in module asyncio)
create_static_lib() (distutils.ccompiler.CCompiler method)
create_stats() (profile.Profile method)
create_subprocess_entry() (in module subprocess)
create_task() (asyncio.loop method)
create_tree() (in module zipfile)
create_tree() (in module shutil)
create_unicode_buffer() (ctypes.create_unix_connection() (asyncio.loop method)
create_unix_server() (asyncio.loop method)
create_version (zipfile.ZipInfo attribute)
createAttribute() (xml.dom.Document method)
createAttributeNS() (xml.dom.Document method)
createComment() (xml.dom.Document method)
createDocument() (xml.dom.DOMImplementation)
createDocumentType() (xml.dom.DOMImplementation)
createElement() (xml.dom.Document method)
createElementNS() (xml.dom.Document method)
createfilehandler() (tkinter.Widget.tk method)
CreateKey() (in module winreg)
CreateKeyEx() (in module winreg)
createLock() (logging.NullHandler method)
createProcessingInstruction() (xml.dom.Document method)
CreateRecord() (msilib.createSocket() (logging.handlers)
createTextNode() (xml.dom.Document method)
CrncyStr (in module locale)
cross() (in module audioop)
crypt (in module

xml.sax.xmlreader.IncrementalParser method)
(zipfile.ZipFile method)
close_connection
(http.server.BaseHTTPRequestHandler attribute)
close_when_done() (asynchat.async_chat method)
closed (http.client.HTTPResponse attribute)
(io.IOBase attribute)
(mmap.mmap attribute)
(ossaudiodev.oss_audio_device attribute)
(select.devpoll attribute)
(select.epoll attribute)
(select.kqueue attribute)
CloseKey() (in module winreg)
closelog() (in module syslog)
closerange() (in module os)
closing() (in module contextlib)
clrtobot() (curses.window method)
clrtoeol() (curses.window method)
cmath (module)
cmd
   module
Cmd (class in cmd)
cmd (module)
   (subprocess.CalledProcessError attribute)
   (subprocess.TimeoutExpired attribute)
cmdloop() (cmd.Cmd method)
cmdqueue (cmd.Cmd attribute)
cmp() (in module filecmp)
code (module)
  (SystemExit attribute)
  (urllib.error.HTTPError attribute)
  (xml.etree.ElementTree.ParseError attribute)
  (xml.parsers.expat.ExpatError attribute)
code object, [1], [2], [3]
code_info() (in module dis)
CodecInfo (class in codecs)
Codecs
  decode
  encode
codecs (module)
coded_value (http.cookies.Morsel attribute)
codeop (module)
codepoint2name (in module html.entities)
codes (in module xml.parsers.expat.errors)
CODESET (in module locale)
CodeType (in module types)
coding
  style
  coercion
  col_offset (ast.AST attribute)
collapse_addresses() (in module ipaddress)
collapse_rfc2231_value() (in module email.utils)
collect() (in module gc)
collect_incoming_data()
  (asyncio.async_chat method)
Collection (class in collections.abc)
  (class in typing)
collections (module)
current_process()
current_task() (asyncio.Task method)
current_thread() (in module asyncio)
CurrentByteIndex (xml.parsers.expat.xmlparser attribute)
CurrentColumnNumber (xml.parsers.expat.xmlparser attribute)
currentframe() (in module asyncio)
curs_set() (in module curses)
curses (module)
curses.ascii (module)
curses.panel (module)
curses.textpad (module)
Cursor (class in sqlite3)
cursyncup() (curses.window method)
customize_compiler() (in module distutils.sysconfig)
Cut
cwd() (ftpplib.FTP method)
cycle() (in module collections.abc)
Cyclic Redundancy Check
collections.abc (module)
colno (json.JSONDecodeError attribute)
(re.error attribute)
Index – D

D_FMT (in module locale)
D_T_FMT (in module locale)
daemon (multiprocessing.Process attribute)
            (threading.Thread attribute)
dangling
else
data
    packing binary
    tabular
    type
    type, immutable
Data (class in plistlib)
data (collections.UserDict attribute)
    (collections.UserList attribute)
    (collections.UserString attribute)
    (select.kevent attribute)
    (selectors.SelectorKey attribute)
    (urllib.request.Request attribute)
    (xml.dom.Comment attribute)
    (xml.dom.ProcessingInstruction attribute)
    (xml.dom.Text attribute)
    (xmlrpc.client.Binary attribute)
data()
(xml.etree.ElementTree.TreeBuilder method)
data_open() (urllib.request.DataHandler method)
data_received() (asyncio.Protocol method)

Differ (class in difflib)
difference() (frozenset method)
difference_update() (frozenset method)
difflib (module)
digest() (hashlib.hash method)
            (hashlib.shake method)
            (hmac.HMAC method)
            (hmac.HMAC method)
digest_size (hmac.HMAC method)
digit() (in module urllib)
digits (in module string)
dir() (built-in function)
    (ftplib.FTP method)
dircmp (class in filecmp)
directory
    changing
    creating
    deleting, [1]
    site-packages
    traversal, [1]
    walking, [1]
Directory (class in os)
directory
    compileall command
    directory_created()
    DirEntry (class in os)
    DirList (class in tkinter)
dirname() (in module os)
database

Unicode
DatabaseError
databases
dataclass() (in module dataclasses)
dataclasses (module)
datagram_received()
 (asyncio.DatagramProtocol method)
DatagramHandler (class in logging.handlers)
DatagramProtocol (class in asyncio)
DatagramRequestHandler (class in socketserver)
DatagramTransport (class in asyncio)
DataHandler (class in urllib.request)
date (class in datetime)
date() (datetime.datetime method)
 (ntplib.NNTP method)
date_time (zipfile.ZipInfo attribute)
date_time_string()
 (http.server.BaseHTTPRequestHandler method)
DateHeader (class in email.headerregistry)
datetime (class in datetime)
DateTime (class in xmlrpc.client)
datetime
 (email.headerregistry.DateHeader attribute)
 (module)
datum
day (datetime.date attribute)
 (datetime.datetime attribute)
day_abbr (in module calendar)

DirSelectBox (class in tkinter.tix)
DirSelectDialog (class
DirsOnSysPath (class
DirTree (class in tkinter)
dis (module)
dis() (dis.Bytecode method)
 (in module dis)
disable (pdb command)
disable() (bdb.Breakpoint method)
 (in module faulthandler)
disable_gc() (in module gc)
disable_interspersed_args()
 (optparse.OptionParser method)
DisableReflectionKey()
 (module)
disassemble() (in module dis)
discover()
 (unittest.TestLoader method)
disk_usage() (in module shutil)
dispatch_call()
 (bdist.egg.distutils.bdist_egg)
dispatch_exception
 (bdist.egg.distutils.bdist_egg)
dispatch_line()
 (bdist.egg.distutils.bdist_egg)
dispatch_return()
 (bdist.egg.distutils.bdist_egg)
dispatch_table (pickle.Pickler attribute)
dispatcher (class in

mailbox.MH method)
dispatch_buffers ()
 (mailbox.Mailbox method)
disco() (in module urllib.request)
discover() (unittest.TestLoader method)
disk_usage() (in module

mailbox.Mailbox method)
day_name (in module calendar)
daylight (in module time)
Daylight Saving Time
DbfilenameShelf (class in shelve)
dbm (module)
dbm.dumb (module)
dbm.gnu
    module, [1]
dbm.gnu (module)
dbm.ndbm
    module, [1]
dbm.ndbm (module)
dcgettext() (in module locale)
deallocation, object
debug (imaplib.IMAP4 attribute)
DEBUG (in module re)
debug (shlex.shlex attribute)
    (zipfile.ZipFile attribute)
debug() (in module doctest)
    (in module logging)
    (logging.Logger method)
    (pipes.Template method)
    (unittest.TestCase method)
    (unittest.TestSuite method)
DEBUG_BYTECODE_SUFFIXES (in module importlib.machinery)
DEBUG_COLLECTABLE (in module gc)
DEBUG_LEAK (in module gc)
debug_print()
    (distutils.ccompiler.CCompiler method)
DEBUG_SAVEALL (in module gc)
debug_src() (in module doctest)
DEBUG_STATS (in module gc)
DEBUG_UNCOLLECTABLE (in module gc)
gc)                  distutils.command.config
debugger, [1], [2]  distutils.command.install
c             configuration file  distutils.command.install_data
debugging                                  distutils.command.install_headers
c           CGI                                   distutils.command.install_lib
c                assertions                           distutils.command.install_scripts
debuggingServer (class in smtpd)           distutils.command.register
debuglevel (http.client.HTTPResponse        distutils.command.sdist
attribute)                                   distutils.core
DebugRunner (class in doctest)              distutils.debug
Decimal (class in decimal)                  distutils.dep_util
decimal (module)                             distutils.dir_util
decimal literal                               distutils.dist
decimal() (in module unicodedata)           distutils.errors
DecimalException (class in decimal)         distutils.extension
decode                                       distutils.fancy_getopt
codecs                                         distutils.file_util
decode (codecs.CodecInfo attribute)          distutils.filelist
decode() (bytearray method)              distutils.log
          (bytes method)                           distutils.msvccompilers
          (codecs.Codec method)                      distutils.spawn
          (codecs.IncrementalDecoder method)       distutils.sysconfig
          (in module base64)                          distutils.text_file
          (in module codecs)                           distutils.unixcompilers
          (in module quopri)                           distutils.util
          (in module uu)                              distutils.version
          (json.JSONDecoder method)                    DISTUTILS_DEBUG
          (xmlrpc.client.Binary method)                 divide()
          (xmlrpc.client.DateTime method)               divide_int()
decode_header() (in module email.header)    division
          (in module nntplib)                                 DivisionByZero (class
decode_params() (in module email.utils)    built-in function

decode_rfc2231() (in module email.utils)
decode_source() (in module importlib.util)
decodebytes() (in module base64)
DecodedGenerator (class in email.generator)
decodestring() (in module base64)
    (in module quopri)
decomposition() (in module unicodedata)
decompress() (bz2.BZ2Decompressor method)
    (in module bz2)
    (in module gzip)
    (in module lzma)
    (in module zlib)
    (lzma.LZMADecompressor method)
    (zlib.Decompress method)
decompressobj() (in module zlib)
decorator
DEDENT (in module token)
DEDENT token, [1]
dedent() (in module textwrap)
deprecated() (in module copy)
def
    statement
def_prog_mode() (in module curses)
def_shell_mode() (in module curses)
default
    parameter value
default (in module email.policy)
DEFAULT (in module unittest.mock)
default (inspect.Parameter attribute)
    (optparse.Option attribute)
default() (cmd.Cmd method)
    (json.JSONEncoder method)
divmod() (built-in function)
    (decimal.Context method)
DllCanUnloadNow() (ctypes method)
DllGetClassObject() (ctypes method)
dllhandle (in module sys)
dngettext() (in module gettext)
do_clear() (bdb.Bdb method)
    (in module bdb)
    (in module curses)
do_command() (cursor command method)
do_GET() (http.server.SimpleHTTPRequestHandler method)
do_handshake() (ssl.SSLSocket method)
do_HEAD() (http.server.SimpleHTTPRequestHandler method)
do_POST() (http.server.CGIHTTPRequestHandler method)
doc (json.JSONDecoder)
doc_header (cmd.Command)
DocCGIXMLRPCRequestHandler (class in xmlrpc.server)
DocFileSuite() (in module doctest)
doCleanups() (unittest.TestCase method)
docmd() (smtplib.SMTP method)
    (doctest.DocTest attribute)
docstrings, [1]
doctest (module)
doctest (class in doctest)
DocTestFailure
DocTestFinder (class in doctest)
DocTestParser (class in doctest)
DocTestRunner (class in doctest)
DocTestSuite() (in module doctest)
DEFAULT_BUFFER_SIZE (in module io)
default_bufsize (in module xml.dom.pulldom)
default_exception_handler() (asyncio.loop method)
default_factory (collections.defaultdict attribute)
DEFAULT_FORMAT (in module tarfile)
DEFAULT_IGNORES (in module filecmp)
default_open() (urllib.request.BaseHandler method)
DEFAULT_PROTOCOL (in module pickle)
default_timer() (in module timeit)
DefaultContext (class in decimal)
DefaultCookiePolicy (class in http.cookiejar)
defaultdict (class in collections)
DefaultDict (class in typing)
DefaultEventLoopPolicy (class in asyncio)
DefaultHandler() (xml.parsers.expat.xmlparser method)
DefaultHandlerExpand() (xml.parsers.expat.xmlparser method)
defaults() (configparser.ConfigParser method)
DefaultSelector (class in selectors)
defaultTestLoader (in module unittest)
defaultTestResult() (unittest.TestCase method)
defects (email.headerregistry.BaseHeader attribute)
doctype() (xml.etree.Element method)
documentation (xml.etree.Element method)
documentation generation online
documentation string
documentation string
documentElement (xml.etree.ElementTree.TreeBuilder method)
DocXMLRPCRequestHandler (class in xmlrpc.server)
DocXMLRPCServer (class in xmlrpc.server)
domain (email.headerregistry.Address attribute)
(tracemalloc.DomainFilter attribute)
(tracemalloc.Filter attribute)
(tracemalloc.Trace attribute)
domain_initial_dot (http.cookiejar.Cookie attribute)
domain_return_ok() (http.cookiejar.CookiePolicy method)
domain_specified (http.cookiejar.Cookie attribute)
DomainFilter (class in tracemalloc)
DomainLiberal (http.cookiejar.DefaultCookiePolicy attribute)
DomainRFC2965Match (http.cookiejar.DefaultCookiePolicy attribute)
DomainStrict (http.cookiejar.DefaultCookiePolicy attribute)
DomainStrictNoDot (http.cookiejar.DefaultCookiePolicy attribute)
DomainStrictNonDomain (http.cookiejar.DefaultCookiePolicy attribute)
(email.message.EmailMessage attribute)
(email.message.Message attribute)
define_macro()
(distutils.ccompiler.CCompiler method)
definition
class, [1]
function, [1]
defpath (in module os)
DefragResult (class in urllib.parse)
DefragResultBytes (class in urllib.parse)
degrees() (in module math)
   (in module turtle)
del
   statement, [1], [2], [3]
del_param()
(email.message.EmailMessage method)
   (email.message.Message method)
delattr() (built-in function)
delay() (in module turtle)
delay_output() (in module curses)
delayload  (http.cookiejar.FileCookieJar attribute)
delch() (curses.window method)
dele() (poplib.POP3 method)
delete() (ftplib.FTP method)
   (imaplib.IMAP4 method)
   (tkinter.ttk.Treeview method)
DELETE_ATTR (opcode)
DELETE_DEREF (opcode)
DELETE_FAST (opcode)
DELETE_GLOBAL (opcode)
DELETE_NAME (opcode)
DELETE_SUBSCR (opcode)

DOMEventStream (class)
DOMException
DomstringSizeErr
done() (asyncio.Future method)
   (asyncio.Task method)
   (concurrent.futures.Future method)
   (in module turtle)
   (xdrlib.Unpacker method)
DONT_ACCEPT_B
doctest)
DONT_ACCEPT_T
doctest)
dont_write_bytecode
doRollover()
   (logging.handlers.R
   (logging.handle
   method)
DOT (in module tok
dot() (in module turi
DOTALL (in module 
doublequote (csv.D
DOUBLESLASH (in
DOUBLESASHEQ
DOUBLESTAR (in
DOUBLESTAREQ

doupdate() (in mod

down (pdb commar
down() (in module t
drain() (asyncio.Str
drop_whitespace
attribute)
dropwhile() (in mod
dst() (datetime.date
   (datetime.time
deleteacl() (imaplib.IMAP4 method)
deletefilehandler() (tkinter.Widget.tk method)
DeleteKey() (in module winreg)
DeleteKeyEx() (in module winreg)
deleteln() (curses.window method)
deleteMe() (bdb.Breakpoint method)
DeleteValue() (in module winreg)
deletion
  attribute
  target
  target list
delimiter (csv.Dialect attribute)
delimiters
delitem() (in module operator)
deliver_challenge() (in module multiprocessing.connection)
delocalize() (in module locale)
demo_app() (in module wsgiref.simple_server)
denominator (fractions.Fraction attribute)
  (numbers.Rational attribute)
DeprecationWarning
deque (class in collections)
Deque (class in typing)
dequeue()
  (logging.handlers.QueueListener method)
DER_cert_to_PEM_cert() (in module ssl)
derwin() (curses.window method)
DES
  cipher
description (sqlite3.Cursor attribute)
description() (nntplib.NNTP method)
descriptions() (nntplib.NNTP method)
**descriptor**
dest (optparse.Option attribute)
destructor, [1]
detach() (io.BufferedIOBase method)
   (io.TextIOBase method)
   (socket.socket method)
   (tkinter.ttk.Treeview method)
   (weakref.finalize method)
Detach() (winreg.PyHKEY method)
DETACHED_PROCESS (in module subprocess)
detect_api_mismatch() (in module test.support)
detect_encoding() (in module tokenize)
detect_language()
   (distutils.ccompiler.CCompiler method)
deterministic profiling
device_encoding() (in module os)
devnull (in module os)
DEVNULL (in module subprocess)
devpoll() (in module select)
DevpollSelector (class in selectors)
dgettext() (in module gettext)
   (in module locale)
Dialect (class in csv)
dialect (csv.csvreader attribute)
   (csv.csvwriter attribute)
Dialog (class in msilib)
dict (2to3 fixer)
   (built-in class)
Dict (class in typing)
dict()
   (multiprocessing.managers.SyncManager method)
dictConfig() (in module logging.config)

dictionary
  display
  object, [1], [2], [3], [4], [5], [6], [7]
  type, operations on
dictionary view
  DictReader (class in csv)
  DictWriter (class in csv)
diff_bytes() (in module difflib)
diff_files (filecmp.dircmp attribute)
Index – E

- e
  - in numeric literal
- e (in module cmath)
- E2BIG (in module errno)
- EACCES (in module errno)
- EADDRINUSE (in module errno)
- EADDRNOTAVAIL (in module errno)
- EADV (in module errno)
- EAFNOSUPPORT (in module errno)
- EAGAIN (in module errno)
- EALREADY (in module errno)
- east_asian_width() (in module unicodedata)
- EBADE (in module errno)
- EBADF (in module errno)
- EBADFD (in module errno)
- EBADMSG (in module errno)
- EBADR (in module errno)
- EBADRQC (in module errno)
- EBADSLT (in module errno)
- EBFONT (in module errno)
- EBUSY (in module errno)
- ECHILD (in module errno)
- eq() (in module curses)
- echo() (curses.window method)
- ECHRNG (in module errno)
- ECOMM (in module errno)
- ECONNABORTED (in module errno)
- ECONNREFUSED (in module errno)
- ECONNRESET (in module errno)
- EOFError (built-in exception)
- EOFError (asyncio.BufferedProtocol method)
- EOFError (asyncio.Protocol method)
- EOFError (email.message.EmailMessage attribute)
- EOFError (email.message.Message attribute)
- EPIPE (in module errno)
- epoch
- epilogue
- EPROTONOSUPPORT (in module errno)
- EPROTO (in module errno)
- EPROTO (selectors)
- EPROTO (epoll()) (in selectors)
- EPROTONOSUPPORT (in module errno)
- EPROTOTYPE (in module errno)
- ERA (in module locale)
- ERA_D_FMT (in module locale)
- ERA_D_T_FMT (in module locale)
- ERA_T_FMT (in module locale)
- ERANGE (in module errno)
- erase() (curses.window method)
- erasechar()
- ERETRY
- EREMOTE
EDEADLK (in module errno)  EREMOTE
EDEADLOCK (in module errno)  ERESTAR
EDESTADDRREQ (in module errno)  erf() (in module math)
edit() (curses.textpad.Textbox method)  erfc() (in module math)
EDOM (in module errno)  EROFS (in module(errno)
EDOTDOT (in module errno)  ERR (in module(errno)
EDQUOT (in module errno)  errcheck (ctypes.
EEXIST (in module errno)  errcode (xmlrpc
EFAULT (in module errno)  errormsg (xmlrpc
EFBIG (in module errno)  errno
effective() (in module bdb)  modul
ehlo() (smtplib.SMTP method)  errno (mod
ehlo_or_helo_if_needed() (smtplib.SMTP method)  (OSError
EHOSTDOWN (in module errno)  Error, [1], [11], [12]
EHOSTUNREACH (in module errno)  error, [1], [11], [12],
EIDRM (in module errno)  error handl
EILSEQ (in module errno)  error() (arg
EINPROGRESS (in module errno)  (in mo
EINTR (in module errno)  (loggir
EINVAL (in module errno)  (urllib.
EIO (in module errno)  (xml.s
EISCONN (in module errno)  error_body
EISDIR (in module errno)  attribute)
EISNAM (in module errno)  error_head
EL2HLT (in module errno)  attribute)
EL2NSYNC (in module errno)  error_leade
EL3HLT (in module errno)  error_mess
(health
EL3RST (in module errno)  (http.serve
Element (class in xml.etree.ElementTree)  (http.serve
element_create() (tkinter.ttk.Style method)  attribute)
element_names() (tkinter.ttk.Style method)  error_lead
element_options() (tkinter.ttk.Style method)  error_mess
ElementDeclHandler()  (http.serve
(xml.parsers.expat.xmlparser method)  attribute)
email.utils (module)
EmailMessage (class in email.message)
EmailPolicy (class in email.policy)
EMFILE (in module errno)
emit() (logging.FileHandler method)
(logging.Handler method)
(logging.NullHandler method)
(logging.StreamHandler method)
(logging.handlers.BufferingHandler method)
(logging.handlers.DatagramHandler method)
(logging.handlers.HTTPHandler method)
(logging.handlers.NTEventLogHandler method)
(logging.handlers.QueueHandler method)
(logging.handlers.RotatingFileHandler method)
(logging.handlers.SMTPHandler method)
(logging.handlers.SocketHandler method)
(logging.handlers.SysLogHandler method)
(logging.handlers.TimedRotatingFileHandler method)
(logging.handlers.WatchedFileHandler method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
EMLINK (in module errno)
Empty
empty
list
tuple, [1]
empty (inspect.Parameter attribute)
(inspect.Signature attribute)
empty() (asyncio.Queue method)
(multiprocessing.Queue method)
(multiprocessing.SimpleQueue method)
(queue.Queue method)
end_headers()
(http.server.BaseHTTPRequestHandler method) exec_module()
(importlib.abc.InspectLoader method)
end_paragraph() (formatter.formatter method) (importlib.abc.Loader method)
end_poly() (in module turtle) (importlib.abc.SourceLoader method)
EndCdataSectionHandler()
(xml.parsers.expat.xmlparser method) (importlib.machinery.ExtensionFileLoader method)
EndDoctypeDeclHandler()
(xml.parsers.expat.xmlparser method) EXEC_PREFIX (in module distutils.sysconfig)
endDocument() (xml.sax.handler.ContentHandler method) execfile (2to3 fixer)
endElement() (xml.sax.handler.ContentHandler method) execle() (in module
EndElementHandler())
(xml.parsers.expat.xmlparser method) execle() (in module sys)
endElementNS() (xml.sax.handler.ContentHandler method) executable
endheaders() (http.client.HTTPConnection method) executable_filename()
EXETYPE (in module distutils.ccompiler)
ENDMARKER (in module token) execute()
EndNamespaceDeclHandler() (in module distutils.util)
(xml.parsers.expat.xmlparser method) Execute() (in module
endpos (re.Match attribute) distutils.ccompiler)
endPrefixMapping() Executable (Zip Files)
exectype()
(xml.sax.handler.ContentHandler method) execute() (sqlite3.Connection method)
endswith() (bytearray method) executescript() (sqlite3.Cursor method)
(bytes method)
(str method)
exectype()
endwin() (in module curses) execution
ENETDOWN (in module errno) frame, restric
ENETRESET (in module errno) stack
ENETUNREACH (in module errno) execution
ENFILE (in module errno)
ENOANO (in module errno) ExecutionL
ENOBUFS (in module errno)
ENOCSI (in module errno)  Executor (in concurrent.futures)
ENODATA (in module errno)  execv() (in module os)
ENODEV (in module errno)  execve() (in module os)
ENOENT (in module errno)  execvp() (in module os)
ENOEXEC (in module errno)  execvpe() (in module os)
ENOLCK (in module errno)  ExFileSelectBox (in tkinter.tix)
ENOLINK (in module errno)  EXFULL (in module errno)
ENOMEM (in module errno)  exists() (in pathlib)
ENOMSG (in module errno)  (tkinter.ttk.Treeview method)
ENONET (in module errno)  exit (built-in)
ENOPKG (in module errno)  exit()
ENOPROTOOPT (in module errno)  (argparse.ArgumentParser method)
enqueue() (logging.handlers.QueueHandler method)
enqueue_sentinel() (logging.handlers.QueueListener method)
ensure_directories() (venv.EnvBuilder method)
ensure_future() (in module asyncio)
ensurepip (module)  expand() (in module winreg)
enter() (sched.scheduler method)  expand_node() (xml.dom.pulldom.DOMEventStream method)
enter_async_context() (contextlib.AsyncExitStack method)  expandtabs() (bytes method)
expandtabs() (bytearray method)  expanduser() (pathlib.Path method)
enter_context() (contextlib.ExitStack method)
enterabs() (sched.scheduler method)
entities (xml.dom.DocumentType attribute)
EntityDeclHandler() (xml.parsers.expat.xmlparser method)
entitydefs (in module html.entities)
EntityResolver (class in xml.sax.handler)
Enum (class in enum)
enum (module)
enum_certificates() (in module ssl)
enum_crls() (in module ssl)
enumerate() (built-in function)
    (in module threading)
EnumKey() (in module winreg)
EnumValue() (in module winreg)
EnvBuilder (class in venv)
environ (in module os)
    (in module posix)
environb (in module posix)
environment
    environment variable
        APPDATA
        AUDIODEV
        BROWSER, [1]
        CC
        CFLAGS, [1], [2]
        COLS, [1]
        COLUMNS, [1]
        COMSPEC, [1]
        CPP
        CPPFLAGS
        DISTUTILS_DEBUG
        HOME, [1], [2], [3], [4]
        HOMEDRIVE, [1]
HOME PATH, [1] 
IDLE STARTUP 
KDEDIR 
LANG, [1], [2], [3], [4] 
LANGUAGE, [1] 
LC_ALL, [1] 
LC_MESSAGES, [1] 
LDCXXSHAREDD 
LD FLAGS 
LINES, [1], [2], [3], [4] 
LNAME 
LOGNAME, [1] 
MIXER DEV 
PAGER 
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], 
[11], [12], [13], [14], [15], [16], [17], [18], [19], 
[20], [21], [22], [23], [24], [25], [26], [27], [28], 
[29], [30], [31], [32], [33], [34] 
PATHEXT, [1] 
PLAT 
POSIXLY_CORRECT 
PYTHON*, [1], [2], [3] 
PYTHONASYNCIODEBUG, [1], [2] 
PYTHONBREAKPOINT, [1], [2], [3], [4] 
PYTHONCASEOK, [1] 
PYTHONCOERCECLOCAL E, [1], [2] 
PYTHONDEBUG, [1], [2] 
PYTHONDEVMODE, [1], [2] 
PYTHONDOCS 
PYTHONDONTWRITEBYTECODE, [1], [2], 
[3], [4], [5], [6] 
PYTHONDUMPREFS, [1] 
PYTHONEXECUTABLE 
PYTHONFAULTHANDLER, [1], [2] 

e xtendleft() 
e x tension 
e xtension 
EXTENSION 
importlib.m 
Extension 
e xtensions 
(http. serve 
attribute) 
External D 
e xtension 
e xtension 
Extra (zipfil 
e xt) 
extract() (t 
extract() (tracel 
extract() (zipfil 
(External 
xml.parser 
extract() 
Extra (xml 
ex 
extra (zipfil 
ex 
traceback 
extract() (tracel 
extract() (zipfil 
ex 
ExtractError 
ex 
extractfile() 
ex 
extractall() 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfil 
ex 
extract() (tracel 
extract() (zipfi
PYTHONHASHSEED, [1], [2], [3], [4], [5], [6], [7], [8]
PYTHONHOME, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]
PYTHONINSPECT, [1], [2], [3]
PYTHONIOENCODING, [1], [2], [3], [4], [5], [6], [7]
PYTHONLEGACYWINDOWSFSENCODING, [1], [2], [3]
PYTHONLEGACYWINDOWSSTDIO, [1], [2], [3], [4]
PYTHONMALLOC, [1], [2], [3], [4], [5], [6]
PYTHONMALLOCSTATS, [1]
PYTHONNOUSERSITE, [1], [2], [3], [4]
PYTHONOPTIMIZE, [1], [2]
PYTHONPATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]
PYTHONPROFILEIMPORTTIME, [1], [2]
PYTHONSHOWALLOCCOUNT
PYTHONSHOWREFCOUNT
PYTHONSTARTUP, [1], [2], [3], [4], [5], [6], [7], [8]
PYTHONTHREADDEBUG
PYTHONTRACEMALLOC, [1], [2], [3]
PYTHONUNBUFFERED, [1], [2]
PYTHONUSERBASE, [1], [2], [3]
PYTHONUSERSITE
PYTHONUTF8, [1], [2], [3], [4]
PYTHONVERBOSE
PYTHONWARNINGS, [1], [2], [3], [4], [5], [6], [7], [8]
PYTHON_DOM
SOURCE_DATE_EPOCH, [1], [2], [3], [4]
SSL_CERT_FILE
SSL_CERT_PATH
SystemRoot
TCL_LIBRARY
TEMP
TERM, [1]
TK_LIBRARY
TMP
TMPDIR
TZ, [1], [2], [3], [4], [5]
USER
USERNAME, [1]
USERPROFILE, [1]
USER_BASE
eexec_prefix, [1], [2]
http_proxy, [1], [2]
no_proxy
prefix, [1], [2], [3]
environment variables
deleting
setting
EnvironmentError
Environments
  virtual
EnvironmentVarGuard (class in test.support)
ENXIO (in module errno)
eof (bz2.BZ2Decompressor attribute)
  (lzma.LZMADecompressor attribute)
  (shlex.shlex attribute)
  (ssl.MemoryBIO attribute)
  (zlib.Decompress attribute)
Index – F

f'
    formatted string literal
f'         formatted string literal
f-string, [1]
f_back (frame attribute)
f_builtins (frame attribute)
f_code (frame attribute)
f_contiguous (memoryview attribute)
f_globals (frame attribute)
f_lasti (frame attribute)
f_lineno (frame attribute)
f_locals (frame attribute)
F_LOCK (in module os)
F_OK (in module os)
F_TEST (in module os)
F_TLOCK (in module os)
f_trace (frame attribute)
f_trace_lines (frame attribute)
f_trace_opcodes (frame attribute)
F_ULOCK (in module os)
fabs() (in module math)
factorial() (in module math)
factory() (importlib.util.LazyLoader class method)
fail() (unittest.TestCase method)
FAIL_FAST (in module doctest)
failfast (unittest.TestResult attribute)
failureException (unittest.TestCase attribute)
failures (unittest.TestResult attribute)
FakePath (class in test.support)
False, [1], [2]
false
False (Built-in object)
    (built-in variable)
family (socket.socket attribute)
fancy_getopt()  (in module distutils.fancy_getopt)
FancyGetopt (class in distutils.fancy_getopt)
FancyURLopener (class in urllib.request)
fast (pickle.Pickler attribute)
FastChildWatcher (class in asyncio)
fatalError()  (xml.sax.handler.ErrorHandler method)
Fault (class in xmlrpc.client)
faultCode (xmlrpc.client.Fault attribute)
faulthandler (module)
faultString (xmlrpc.client.Fault attribute)
fchdir() (in module os)
fchmod() (in module os)
fchown() (in module os)
FCICreate() (in module msilib)
fcntl (module)
fcntl() (in module fcntl)
f (selectors.SelectorKey attribute)
f() (in module turtle)
f_count() (in module test.support)
fdatasync() (in module os)
fopen() (in module os)
Feature (class in msilib)
feature_external_ges (in module xml.sax.handler)
feature_external_pes (in module xml.sax.handler)
feature_namespace_prefixes (in module
xml.sax.handler)

feature_namespaces   (in module xml.sax.handler)

feature_string_interning   (in module xml.sax.handler)

feature_validation (in module xml.sax.handler)

feed() (email.parser.BytesFeedParser method)
   (html.parser.HTMLParser method)
   (xml.etree.ElementTree.XMLParser method)
   (xml.etree.ElementTree.XMLPullParser method)
   (xml.sax.xmlreader.IncrementalParser method)

FeedParser (class in email.parser)

fetch() (imaplib.IMAP4 method)
Fetch() (msilib.View method)

fetchall() (sqlite3.Cursor method)
fetchmany() (sqlite3.Cursor method)
fetchone() (sqlite3.Cursor method)

fflags (select.kevent attribute)

Field (class in dataclasses)

field() (in module dataclasses)
field_size_limit() (in module csv)
fieldnames (csv.csvreader attribute)
fields (uuid.UUID attribute)
fields() (in module dataclasses)

file
   .ini
   .pdbrc
   byte-code, [1]
   configuration
   copying
   debugger configuration

fnmatchcase()
focus() (tkinter.ttk.Treeview method)
fold (datetime.datetime attribute)
fold() (email.headerregistry.BaseHeader method)
fold_binary() (email.policy.Compat32 method)
fold() (email.policy.EmailPolicy method)
fold() (email.policy.Policy method)
FOR_ITER (in comprehensions)
for
   in comprehension
   statement

forget() (in module test.support)
fork() (in module os)

ForkingMixIn (class in socketserver)
ForkingTCPServer (class in socketserver)
ForkingUDPServer (class in socketserver)
forkpty() (in module pty)

form
lambda

Form (class in tkinter.tix)
format (memoryview attribute)
format() (built-in function)
large files
mime.types
modes
object, [1]
path configuration
plist
temporary
file (pyclbr.Class attribute)
(pyclbr.Function attribute)
file ...
compileall command line option
file control
UNIX
file name
temporary
file object
io module
open() built-in function
file-like object
FILE_ATTRIBUTE_ARCHIVE (in module stat)
FILE_ATTRIBUTE_COMPRESSED (in module stat)
FILE_ATTRIBUTE_DEVICE (in module stat)
FILE_ATTRIBUTE_DIRECTORY (in module stat)
FILE_ATTRIBUTE_ENCRYPTED (in module stat)
FILE_ATTRIBUTE_HIDDEN (in module stat)
FILE_ATTRIBUTE_INTEGRITY_STREAM (in module stat)
FILE_ATTRIBUTE_NO_SCRUB_DATA (in module stat)
FILE_ATTRIBUTE_NORMAL (in module stat)
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
__str__()
format() (in module logging)
format() (in module logging)
format() (in module pprint)
format() (in module string)
format() (in module traceback)
format() (in module traceback)
format() (in module traceback)
format_datetime() (in module email.utils)
format_exc() (in module traceback)
format_exception() (in module traceback)
format_exception_only() (in module traceback)
format_field() (string.Formatter method)
format_help() (argparse.ArgumentParser method)
format_list() (traceback)
format_map() (str method)
format_stack() (in module traceback)
format_stack_entry() (bdb.Bdb method)
format_string() (in module locale)
format_tb() (in module traceback)
format_usage() (argparse.ArgumentParser method)
FORMAT_VALUE (opcode)
FILE_ATTRIBUTE_OFFLINE (in module stat)
FILE_ATTRIBUTE_READONLY (in module stat)
FILE_ATTRIBUTE_REPARSE_POINT (in module stat)
FILE_ATTRIBUTE_SPARSE_FILE (in module stat)
FILE_ATTRIBUTE_SYSTEM (in module stat)
FILE_ATTRIBUTE_TEMPORARY (in module stat)
FILE_ATTRIBUTE_VIRTUAL (in module stat)
file_created() (built-in function)
file_dispatcher (class in asyncore)
file_open() (urllib.request.FileHandler method)
file_size (zipfile.ZipInfo attribute)
file_wrapper (class in asyncore)
filecmp (module)
fileConfig() (in module logging.config)
FileCookieJar (class in http.cookiejar)
FileEntry (class in tkinter.tix)
FileExistsError
FileFinder (class in importlib.machinery)
FileHandler (class in logging)
file_mode (in module stat)
filename (doctest.DocTest attribute)

formataddr()
formatargspec()
formatargvalues()
formatdate()
FormatError
FormatError()
formatException()
formatmonth()
(calendar.HT)
(calenda)
formatStack()
formatted_string()
Formatter (class in logging)
Formatter (class in string)
formatTime()
formatwarning()
formatyear()
(formatting)
bytearray (%)
bytes (%)
formatting, st
formatwarnin
formatyear() method
field (in module fileinput)
FileIO (class in io)
filelineno() (in module fileinput)
FileLoader (class in importlib.abc)
forward() (in module)
found_terminator()
(http.cookiejar.FileCookieJar attribute)

Fortran contiguous
asynchat.async_chat method
find_library()
find_library_file()
find_loader()
find_longest_match()
find_module()
find_msvcrts()
find_spec()
find_spec()
find_unused_port()
fnd_user_password()
findall()
(xml.etree.ElementTree.ElementTree method)
findCaller() (logging.Logger method)
finder, [1]
    find_spec
Finder (class in importlib.abc)
findfactor() (in module audioop)
findfile() (in module test.support)
findfit() (in module audioop)
finditer() (in module re)
    (re.Pattern method)
findlabels() (in module dis)
findlinestarts() (in module dis)
findmatch() (in module mailcap)
findmax() (in module audioop)
findtext() (xml.etree.ElementTree.Element method)
    (xml.etree.ElementTree.ElementTree method)
finish() (socketserver.BaseRequestHandler method)
finish_request() (socketserver.BaseServer method)
firstChild (xml.dom.Node attribute)

ftp.TLS (class in ftplib)
FTPHandler (class in urllib.request)
ftplib (module)
ft truncate() (in)
Full
full() (asyncio.Queue method)
    (multiprocessing.Queue method)
full_url
    (queue.Queue method)
func (functools.partial method)
    (2to3 fixer)
funcattrs (2to3 fixer)
function
    (functools.partial method)
    (xml.etree.ElementTree.ElementTree method)
    (xml.etree.ElementTree.ElementTree method)
    (socketserver.BaseRequestHandler method)
    (socketserver.BaseServer method)
Function (class)
    function annotation
    anonymous
    argument
call, [1], [1]
call, user-defined
definition
generator
name, [1]
object, [1]
user-defined
FunctionTestCase (class in unittest)
FunctionType (in module types)
functools (module)
funny_files (filecmp.dircmp attribute)
future

future (2to3 fixer)
Future (class in asyncio)
FutureWarning
fwalk() (in module os)
Index – G

G.722

gaierror

gamma() (in module math)
gammavariate() (in module random)
garbage (in module gc)
garbage collection, [1]
gather() (curses.textpad.Textbox method)

gauss() (in module random)
gc (module)
gc__done (C function)
gc__start (C function)
gc_collect() (in module test.support)
gcd() (in module fractions)
  (in module math)
ge() (in module operator)
gen_lib_options() (in module distutils.ccompiler)
gen_preprocess_options() (in module distutils.ccompiler)
gen_uuid() (in module msilib)
generate_help()
  (distutils.fancy_getopt.FancyGetopt method)
generator, [1]
    expression
    function, [1], [2]
    iterator, [1]
    object, [1], [2]
Generator (class in collections.abc)
  (class in email.generator)
  (class in typing)
generator expression, [1]
generator iterator

columnNumber()

getColumnNumber()

getcomments()

getcompname()

getcompty()

getcontext()

getcoroutinelocals()

getcoroutinestate()

getcwd()

getcwdb() (2to3 fixer)

getcwd()

getctime()

getcwdb() (xml.sax.xmlreader)

getdefaultencoding()

getdefaultlocale()

getdefaulttimeout()

getdlopenflags()

getdoc()

getDOMImpl()

getdlopenflags()

getegid() (xml.sax.xmlreader)

getEffectiveLevel()

getegid() (xml.sax.xmlreader)
get_children() (symtable.SymbolTable method)  getint() (xml.parser.expat.xmlparser method)
(ttkinter.ttk.Treeview method)  getChildren()
get_ciphers() (ssl.SSLContext method)  GetInteger()
get_clock_info() (in module time)  get_close_matches() (in module difflib)
get_close_matches() (in module difflib)  get_completion_type() (in module readline)
get_close_matches() (in module difflib)  get_config_h_filename() (in module distutils.sysconfig)
get_code() (importlib.abc.InspectLoader method)  get_code() (importlib.abc.InspectLoader method)
(importlib.abc.SourceLoader method)  get_iterator() (xml.etree.Element method)
(importlib.machinery.ExtensionFileLoader method)  getiterator() (xml.etree.Element method)
(importlib.machinery.SourcelessFileLoader method)  (importlib.machinery.SourcelessFileLoader method)
(zipimport.zipimporter method)  (zipimport.zipimporter method)
get_completer() (in module readline)  getcompleter() (in module readline)
get_completer_delims() (in module readline)  get_completion_type() (in module readline)
get_completion_type() (in module readline)  get_config_h_filename() (in module distutils.sysconfig)
get_config_var() (in module distutils.sysconfig)  get_config_var() (in module distutils.sysconfig)
get_config_vars() (in module distutils.sysconfig)  get_config_vars() (in module distutils.sysconfig)
get_content() (email.contentmanager.ContentManager method)  get_content() (email.contentmanager.ContentManager method)
(email.message.EmailMessage method)  (email.message.EmailMessage method)
get_content_charset() (email.message.EmailMessage method)  (email.message.EmailMessage method)
(getloadavg() (in module os)  (getloadavg() (in module os)
getlocale() (in module locale)  getlocale() (in module locale)
get_logger() (in module logging)  get_logger() (in module logging)
get_logger_class() (in module logging)  get_logger_class() (in module logging)
get_login() (in module os)  get_login() (in module os)
get_log_record_factory() (in module logging)  get_log_record_factory() (in module logging)
getmark() (sunau)  getmark() (sunau)
getmarkers() (sunau)  getmarkers() (sunau)
getmaxyx() (wave)  getmaxyx() (wave)
getmember() (tarfile.TarFile method)  getmember() (tarfile.TarFile method)
get_from() (mailbox.mboxMessage method)  (mailbox.MMDFMessage method)
get_full_url() (urllib.request.Request method)
get_globals() (symtable.Function method)
get_grouped_opcodes() (difflib.SequenceMatcher method)
get_handle_inheritable() (in module os)
get_header() (urllib.request.Request method)
get_history_item() (in module readline)
get_history_length() (in module readline)
get_id() (symtable.SymbolTable method)
get_ident() (in module _thread)
get_identifiers() (symtable.SymbolTable method)
get_importer() (in module pkgutil)
get_info() (mailbox.MaildirMessage method)
get_inheritable() (in module os)
get_instructions() (in module dis)
get_interpreter() (in module zipapp)
GET_ITER (opcode)
get_key() (selectors.BaseSelector method)
get_labels() (mailbox.Babyl method)
get_last_error() (in module ctypes)
get_line_buffer() (in module readline)
get_lineno() (symtable.SymbolTable method)
get_loader() (in module pkgutil)
get_locals() (symtable.Function method)
get_logger() (in module multiprocessing)
get_loop() (asyncio.Future method)
get_magic() (in module imp)
get_makefile_filename() (in module
getpwall() (in module pwd)
getpwknam()
getpwuid()
getQName
getQName
getquota()
getquotaroot()
getrandbits
getrandom
getreader()
getreces
getrefcount
getresgid()
getresidu
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresidu
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
getresgid()
distutils.sysconfig
   (in module sysconfig)
get_map() (selectors.BaseSelector method)
get_matching_blocks() (difflib.SequenceMatcher method)
get_message() (mailbox.Mailbox method)
get_method() (urllib.request.Request method)
get_methods() (symtable.Class method)
get_mixed_type_key() (in module ipaddress)
get_name() (symtable.Symbol method)
   (symtable.SymbolTable method)
get_namespace() (symtable.Symbol method)
get_namespaces() (symtable.Symbol method)
get_nonstandard_attr() (http.cookiejar.Cookie method)
get_nowait() (asyncio.Queue method)
   (multiprocessing.Queue method)
   (queue.Queue method)
   (queue.SimpleQueue method)
get_object_traceback() (in module tracemalloc)
get_objects() (in module gc)
get_opcodes() (difflib.SequenceMatcher method)
get_option() (optparse.OptionParser method)
get_option_group() (optparse.OptionParser method)
get_option_order()
   (distutils.fancy_getopt.FancyGetopt method)
get_original_stdout() (in module test.support)
get_osfhandle() (in module msvcrt)
get_output_charset() (email.charset.Charset method)
get_param() (email.message.Message method)
get_parameters() (symtable.Function method)
get_params() (email.message.Message method)
get_source()  (importlib.abc.InspectLoader method)
    (importlib.abc.SourceLoader method)
    (importlib.machinery.ExtensionFileLoader method)
    (importlib.machinery.SourcelessFileLoader method)
    (zipimport.zipimporter method)
get_special_folder_path() (built-in function)
get_stack() (asyncio.Task method)
    (bdb.Bdb method)
get_start_method() (in module multiprocessing)
get_starttag_text() (html.parser.HTMLParser method)
global
get_stats() (in module gc)
global
get_stderr() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
global
get_stdin() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
global
get_string() (mailbox.Mailbox method)
global
get_subdir() (mailbox.MaildirMessage method)
global
get_suffixes() (in module imp)
global
get_symbols() (symtable.SymbolTable method)
global
get_tag() (in module imp)
global
get_task_factory() (asyncio.loop method)
glob
get_terminal_size() (in module os)
    (in module shutil)
global
get_terminator() (asynchat.async_chat method)
glob
get_threshold() (in module gc)
glob
get_token() (shlex.shlex method)
glob
get_traceback_limit() (in module tracemalloc)
glob
get_traced_memory() (in module tracemalloc)
glob
get_tracemalloc_memory() (in module tracemalloc)
get_type() (symtable.SymbolTable method)
get_type_hints() (in module typing)
get_unixfrom() (email.message.EmailMessage method)
(get_email.message.Message method)
get_unpack_formats() (in module shutil)
get_usage() (optparse.OptionParser method)
get_value() (string.Formatter method)
get_version() (optparse.OptionParser method)
get_visible() (mailbox.BabylMessage method)
get_wch() (curses.window method)
get_write_buffer_limits() (asyncio.WriteTransport method)
get_write_buffer_size() (asyncio.WriteTransport method)
GET_YIELD_FROM_ITER (opcode)
getacl() (imaplib.IMAP4 method)
getaddresses() (in module email.utils)
getaddrinfo() (asyncio.loop method)
(in module socket)
getallocatedblocks() (in module sys)
getandroidapilevel() (in module sys)
getannotation() (imaplib.IMAP4 method)
getargspec() (in module inspect)
getargvalues() (in module inspect)
getatime() (in module os.path)
getattr() (built-in function)
getattr_static() (in module inspect)
getAttribute() (xml.dom.Element method)
getAttributeNode() (xml.dom.Element method)
getAttributeNodeNS() (xml.dom.Element method)
group() (nn)
groupdict() (re.Match method)
groupby() (itertools method)
groupdict() (pathlib.Path method)
grouping (re.Pattern attribute)
groupindex (re.Pattern attribute)
groups() (re.Pattern method)
groups() (re.Match method)
guess_all_extensions() (mimetypes method)
guess_extension() (mimetypes method)
guess_scheme() (wsgiref.util method)
guess_type() (mimetypes method)
GUI gzip (module)
GzipFile (class)
GetBase() (xml.parsers.expat.xmlparser method)
getbegyx() (curses.window method)
getbkgd() (curses.window method)
getblocking() (socket.socket method)
getboolean() (configparser.ConfigParser method)
getbuffer() (io.BytesIO method)
getByteStream() (xml.sax.xmlreader.InputSource method)
getcallargs() (in module inspect)
getcanvas() (in module turtle)
getcapabilities() (nntplib.NNTP method)
getcaps() (in module mailcap)
getch() (curses.window method)
    (in module msvcrt)
getCharacterStream()
(xml.sax.xmlreader.InputSource method)
getche() (in module msvcrt)
getcheckinterval() (in module sys)
getChild() (logging.Logger method)
getchildren() (xml.etree.ElementTree.Element method)
getclasstree() (in module inspect)
getclosurevars() (in module inspect)
GetColumnInfo() (msilib.View method)
Index – H

halfdelay() (in module curses)  heappush()
Handle (class in asyncio)  heappushpop
handle an exception  heapq (module)
handle()  (http.server.BaseHTTPRequestHandler method)
  (logging.Handler method)
  (logging.Logger method)
  (logging.NullHandler method)
  (logging.handlers.QueueListener method)
  (socketserver.BaseRequestHandler method)
  (wsgiref.simple_server.WSGIRequestHandler method)
handle_accept()  (asyncore.dispatcher method)
handle_accepted()  (asyncore.dispatcher method)
handle_charref()  (html.parser.HTMLParser method)
handle_close()  (asyncore.dispatcher method)
handle_comment()  (html.parser.HTMLParser method)
handle_connect()  (asyncore.dispatcher method)
handle_data()  (html.parser.HTMLParser method)
handle_decl()  (html.parser.HTMLParser method)
handle_defect()  (email.policy.Policy method)
handle_endtag()  (html.parser.HTMLParser method)
handle_entityref()  (html.parser.HTMLParser method)
handle_error()  (asyncore.dispatcher method)
  (socketserver.BaseServer method)
handle_expect_100()  (http.server.BaseHTTPRequestHandler method)
heappush()
heappushpop
heapq (module)
heapreplacer
heclo() (smt)
handle_expt() (asyncore.dispatcher method)
handle_one_request()
(http.server.BaseHTTPRequestHandler method)
handle_pi() (html.parser.HTMLParser method)
handle_read() (asyncore.dispatcher method)
handle_request() (socketserver.BaseServer method)
(xmlrpc.server.CGIHTTPRequestHandler method)
handle_startendtag() (html.parser.HTMLParser method)
handle_starttag() (html.parser.HTMLParser method)
handle_timeout() (socketserver.BaseServer method)
handle_write() (asyncore.dispatcher method)
handleError() (logging.Handler method)
(handler, exception
Handler (class in logging)
handler() (in module cgitb)
harmonic_mean() (in module statistics)
HAS_ALPN (in module ssl)
has_children() (symtable.SymbolTable method)
has_colors() (in module curses)
HAS_ECDH (in module ssl)
has_exec() (symtable.SymbolTable method)
has_extn() (smtplib.SMTP method)
has_function() (distutils.ccompiler.CCompiler method)
has_header() (csv.Sniffer method)
(haslib.request.Request method)
has_ic() (in module curses)
handler
exception
Handler (class in logging)
handler() (in module cgitb)
harmonic_mean() (in module statistics)
HAS_ALPN (in module ssl)
has_children() (symtable.SymbolTable method)
has_colors() (in module curses)
HAS_ECDH (in module ssl)
has_exec() (symtable.SymbolTable method)
has_extn() (smtplib.SMTP method)
has_function() (distutils.ccompiler.CCompiler method)
has_header() (csv.Sniffer method)
(haslib.request.Request method)
has_ic() (in module curses)
handler
exception
Handler (class in logging)
handler() (in module cgitb)
harmonic_mean() (in module statistics)
HAS_ALPN (in module ssl)
has_children() (symtable.SymbolTable method)
has_colors() (in module curses)
HAS_ECDH (in module ssl)
has_exec() (symtable.SymbolTable method)
has_extn() (smtplib.SMTP method)
has_function() (distutils.ccompiler.CCompiler method)
has_header() (csv.Sniffer method)
(haslib.request.Request method)
has_ic() (in module curses)
has_Il() (in module curses)
has_ipv6 (in module socket)
has_key (2to3 fixer)
has_key() (in module curses)
has_location  (importlib.machinery.ModuleSpec attribute)
HAS_NEVER_CHECK_COMMON_NAME  (in module ssl)
has_nonstandard_attr()  (http.cookiejar.Cookie method)
HAS_NPN (in module ssl)
has_option() (configparser.ConfigParser method)
        (optparse.OptionParser method)
has_section() (configparser.ConfigParser method)
HAS_SNI (in module ssl)
HAS_SSLv2 (in module ssl)
HAS_SSLv3 (in module ssl)
has_ticket (ssl.SSLSession attribute)
HAS_TLSv1 (in module ssl)
HAS_TLSv1_1 (in module ssl)
HAS_TLSv1_2 (in module ssl)
HAS_TLSv1_3 (in module ssl)
hasattr() (built-in function)
hasAttribute() (xml.dom.Element method)
hasAttributeNS() (xml.dom.Element method)
hasAttributes() (xml.dom.Node method)
hasChildNodes() (xml.dom.Node method)
hascompare (in module dis)
hasconst (in module dis)
hasFeature()  (xml.dom.DOMImplementation method)
hasfree (in module dis)
hash
        built-in function, [1], [2], [3]
host (urllib.
hostmask (ipaddress.
hostname_ (ssl.SSLContext attribute)
hosts (netrc.netrc attribute)
hosts() (ipaddress.
HREQUEST
HRStdError
HStdInput
hStdOutput
hasTicket (ssl.SSLSession attribute)
HAS_TESv1 (in module ssl)
HAS_TLSv1_1 (in module ssl)
HAS_TLSv1_2 (in module ssl)
HAS_TLSv1_3 (in module ssl)
hasattr() (built-in function)
hasAttribute() (xml.dom.Element method)
hasAttributeNS() (xml.dom.Element method)
hasAttributes() (xml.dom.Node method)
hasChildNodes() (xml.dom.Node method)
hascompare (in module dis)
hasconst (in module dis)
hasFeature()  (xml.dom.DOMImplementation method)
hasfree (in module dis)
hash
        built-in function, [1], [2], [3]
HTTP
http (standard module)
http.client (standard module)
HTML, [1]
HTMLCalendar
t(HtmIDiff (class
tHTMLParser
thtonl() (in module
thtons() (in module
tHTTP
http (standard module)
http.client (standard module)
hash character
hash() (built-in function)
**hash-based pyc**
hash.block_size (in module hashlib)
hash.digest_size (in module hashlib)
hash_info (in module sys)
hashable, [1]

Hashable (class in collections.abc)
       (class in typing)

hasHandlers() (logging.Logger method)
hashlib (module)
hasjabs (in module dis)
hasjrel (in module dis)
haslocal (in module dis)
hasname (in module dis)

HAVE_ARGUMENT (opcode)
HAVE_DOCSTRINGS (in module test.support)

HAVE_THREADS (in module decimal)

HCI_DATA_DIR (in module socket)

HCI_FILTER (in module socket)

HCI_TIME_STAMP (in module socket)

head() (ntplib.NNTP method)

Header (class in email.header)
header_encode() (email.charset.Charset method)
header_encode_lines() (email.charset.Charset method)
header_encoding (email.charset.Charset attribute)

header_factory (email.policy.EmailPolicy attribute)
header_fetch_parse() (email.policy.Compat32 method)
       (email.policy.EmailPolicy method)
       (email.policy.Policy method)

header_items() (urllib.request.Request method)
header_max_count()  (email.policy.EmailPolicy method)
   (email.policy.Policy method)
header_offset (zipfile.ZipInfo attribute)
header_source_parse()  (email.policyCompat32 method)
   (email.policy.EmailPolicy method)
   (email.policy.Policy method)
header_store_parse()  (email.policyCompat32 method)
   (email.policy.EmailPolicy method)
   (email.policy.Policy method)
HeaderError
HeaderParseError
HeaderParser (class in email.parser)
HeaderRegistry (class in email.headerregistry)
headers
   MIME, [1]
Headers (class in wsgiref.headers)
headers (http.server.BaseHTTPRequestHandler attribute)
   (urllib.error.HTTPError attribute)
   (xmlrpc.client.ProtocolError attribute)
heading() (in module turtle)
   (tkinter.ttk.Treeview method)
heapify() (in moduleheapq)
heapmin() (in module msvcrth)
heappop() (in moduleheapq)

HTTP_Po
http_proxy
http_response()
http_version attribute)
HTTPBasic
urllib.request
HTTPCon
HTTPCookie
urllib.request
httpd
HTTPEefault
urllib.request
HTTDPdigest
urllib.request
HTTPErrort
HTTPExce
HTTPhand
   (class
HTTPass
HTTPPass
in urllib.request
HTTPPass
urllib.request
HTTPRedi
HTTPRes
https_open method)
HTTPS_Po
Index – I

I (in module re)
I/O control
  POSIX
  UNIX
  buffering, [1]
  tty
iadd() (in module operator)
iand() (in module operator)
iconcat() (in module operator)
id
  built-in function
id (ssl.SSLSession attribute)
id() (built-in function)
  (unittest.TestCase method)
idcok() (curses.window method)
ident (select.kevent attribute)
  (threading.Thread attribute)
identchars (cmd.Cmd attribute)
identifier, [1]
identify()  (tkinter.ttk.Notebook method)
  (tkinter.ttk.Treeview method)
  (tkinter.ttk.Widget method)
identify_column()
  (tkinter.ttk.Treeview method)
identify_element()
  (tkinter.ttk.Treeview method)
identify_region()  (tkinter.ttk.Treeview method)
identify_row()  (tkinter.ttk.Treeview method)
intersection_update() (frozenset method)
IntFlag (class in enum)
intro (cmd.Cmd attribute)
InuseAttributeErr
inv() (in module operator)
InvalidAccessErr
invalidate_caches()  (importlib.abc.MetaPathFinder method)
  (importlib.abc.PathFinder method)
  (importlib.machinery.FileFinder method)
  (importlib.machinery.PathFinder method)
  (in module importlib)
InvalidCharacterErr
InvalidModificationErr
InvalidOperation (class in io)
InvalidStateErr
InvalidStateError
InvalidURL
inversion
invert() (in module operator)
invocation
io
  module
IO (class in typing)
io (module)
io.StringIO
  object
IOBase (class in io)
ioctl() (in module fcntl)
  (socket.socket method)
identity
test
identity of an object
idioms (2to3 fixer)
IDLE, [1]
IDLE_PRIORITY_CLASS (in module subprocess)
IDLESTARTUP
idlok() (curses.window method)
if
    conditional expression
    in comprehensions
    statement, [1]
if_indextoname() (in module socket)
if_nameindex() (in module socket)
if_nametoindex() (in module socket)
ifloordiv() (in module operator)
iglob() (in module glob)
ignorableWhitespace()
(xml.sax.handler.ContentHandler method)
ignore (pdb command)
ignore_errors() (in module codecs)
IGNORE_EXCEPTION_DETAIL (in module doctest)
ignore_patterns() (in module shutil)
IGNORECASE (in module re)
ihave() (ntplib.NNTP method)
IISCGIHandler (class in wsgiref.handlers)
ilshift() (in module operator)
imag (numbers.Complex attribute)
imaginary literal
imap() (multiprocessing.pool.Pool
IOCTL_VM_SOCKETS_MODULE_SOCKET
IOError
ior() (in module operator)
ip (ipaddress.IPv4Interface
   (ipaddress.IPv6Interface
   ip_address() (in module
   ip_interface() (in module
   ip_network() (in module
   ipaddress (module)
ipow() (in module operator)
ipv4_mapped (ipaddress
IPv4Address (class in ip
IPv4Interface (class in ip
IPv4Network (class in ip
IPV6_ENABLED (in module test.support)
IPv6Address (class in ip
IPv6Interface (class in ip
IPv6Network (class in ip
irshift() (in module operator)
is
   operator, [1]
is not
   operator, [1]
is_() (in module operator)
is_absolute() (pathlib.PurePath
   (threading.Thread method)
is_android (in module test.support)
is_assigned() (symtable.Symbol method)
is_attachment() (email.message.EmailMessage method)
is_authenticated() (urllib.request.HTTPPasswordManager method)
method)
IMAP4
    protocol
IMAP4 (class in imaplib)
IMAP4.abort
IMAP4.error
IMAP4.readonly
IMAP4_SSL
    protocol
IMAP4_SSL (class in imaplib)
IMAP4_stream
    protocol
IMAP4_stream (class in imaplib)
imap_unordered()
    (multiprocessing.pool.Pool method)
imaplib (module)
imatmul() (in module operator)
imghdr (module)
immedok() (curses.window method)
immutable
data type
    object, [1], [2]
    sequence types
immutable object
immutable sequence
    object
immutable types
    subclassing
imod() (in module operator)
imp
    module
imp (module)
ImplImporter (class in pkgutil)
impl_detail() (in module test.support)

is_block_device() (pathlib.Path method)
is_blocked() (http.cookiejar.Cookie method)
is_canonical() (decimal.Decimal method)
is_char_device() (pathlib.Path method)
IS_CHARACTER_JUNK
is_check_supported() (in module lzma)
is_closed() (asyncio.loop method)
is_closing() (asyncio.BaseTransport method)
is_dataclass() (in module dataclasses)
is_declared_global() (symtable.Symbol method)
is_dir() (os.DirEntry method)
is_file() (os.DirEntry method)
is_finalizing() (in module sys)
is_finite() (decimal.Decimal method)
is_free() (symtable.Symbol method)
is_global (ipaddress.IPv4Address method)
is_global (ipaddress.IPv6Address method)
is_hop_by_hop() (in module wsgiref.util)
is_imported() (symtable.Symbol method)
is_infinite() (decimal.Decimal method)
is_integer() (float method)
implementation (in module sys)
ImpLoader (class in pkgutil)
import
    hooks
        statement, [1], [2], [3], [4]
import (2to3 fixer)
import hooks
import machinery
**import path**
import__find__load__done (C function)
import__find__load__start (C function)
import_fresh_module() (in module test.support)
IMPORT_FROM (opcode)
import_module() (in module importlib)
    (in module test.support)
IMPORT_NAME (opcode)
IMPORT_STAR (opcode)
**importer**
ImportError
  exception
**importing**
importlib (module)
importlib.abc (module)
importlib.machinery (module)
importlib.resources (module)
importlib.util (module)
imports (2to3 fixer)
imports2 (2to3 fixer)
ImportWarning
ImproperConnectionState
is_jython (in module test)
IS_LINE_JUNK() (in module difflib)
is_linetouched() (curses)
is_link_local (ipaddress.
    (ipaddress.IPv4Net,
    (ipaddress.IPv6Add
    (ipaddress.IPv6Net,
    is_local() (symtable.Symbol
    is_loopback (ipaddress.
    (ipaddress.IPv4Net,
    (ipaddress.IPv6Add
    (ipaddress.IPv6Net,
    is_mount() (pathlib.Path
    is_multicast (ipaddress.
    (ipaddress.IPv4Net,
    (ipaddress.IPv6Add
    (ipaddress.IPv6Net,
    is_multipart() (email.mime.
    (email.mime.Message
    is_namespace() (symtable.
    is_nan() (decimal.Context
    (decimal.Decimal
    is_nested() (symtable.Symbol
    is_normal() (decimal.Context
    (decimal.Decimal
    is_not() (in module operator)
    is_not_allowed()
    (http.cookiejar.DefaultCookiePolicy
    is_optimized() (symtable.Symbol
    is_package() (importlib.
    (importlib.abc.Source	
imul() (in module operator)

in

keyword

operator, [1], [2]

in_dll() (ctypes._CData method)

in_table_a1() (in module stringprep)
in_table_b1() (in module stringprep)
in_table_c11() (in module stringprep)
in_table_c11_c12() (in module stringprep)
in_table_c12() (in module stringprep)
in_table_c21() (in module stringprep)
in_table_c21_c22() (in module stringprep)
in_table_c22() (in module stringprep)
in_table_c3() (in module stringprep)
in_table_c4() (in module stringprep)
in_table_c5() (in module stringprep)
in_table_c6() (in module stringprep)
in_table_c7() (in module stringprep)
in_table_c8() (in module stringprep)
in_table_c9() (in module stringprep)
in_table_d1() (in module stringprep)
in_table_d2() (in module stringprep)
in_transaction (sqlite3.Connection attribute)

inch() (curses.window method)

inclusive

or

inclusive (tracemalloc.DomainFilter attribute)

(tracemalloc.Filter attribute)

Incomplete

IncompleteRead

method)

(method)

(method)

(zipimport.zipimport method)

is_parameter() (symtable Symbol method)

is_private (ipaddress.IPv4Address attribute)

(ipaddress.IPv4Network attribute)

(ipaddress.IPv6Address attribute)

(ipaddress.IPv6Network attribute)

is_python_build() (in module sysconfig)

is_qnan() (decimal.Context method)

(isdecimal.Decimal method)

is_reading() (asyncio.ReadTransport method)

is_referenced() (symtable Symbol method)

(ipaddress.IPv4Address attribute)

(ipaddress.IPv4Network attribute)

(ipaddress.IPv6Address attribute)

(ipaddress.IPv6Network attribute)

is_reserved() (pathlib.PurePath method)

is_resource_enabled() (importlib.abc.ResourceReader method)

(isdecimal.Decimal method)

is_safe (uuid.UUID attribute)

is_serving() (asyncio.Server method)

(isdecimal.Decimal method)

is_set() (asyncio.Event method)

(threading.Event method)

is_signed() (decimal.Decimal method)

is_site_local (ipaddress.IPv4Address attribute)

(ipaddress.IPv4Network attribute)

(ipaddress.IPv6Address attribute)

(ipaddress.IPv6Network attribute)
IncompleteReadError
incr_item(), [1]
increment_lineno() (in module ast)
IncrementalDecoder (class in codecs)
incrementaldecoder
(codecs.CodecInfo attribute)
IncrementalEncoder (class in codecs)
incrementalencoder
(codecs.CodecInfo attribute)
IncrementalNewlineDecoder (class in io)
IncrementalParser (class in xml.sax.xmlreader)
indent (doctest.Example attribute)
INDENT (in module token)
INDENT token
indent() (in module textwrap)
indentation
IndentationError
index operation
index() (array.array method)
(bytarray method)
(bytes method)
(collections.deque method)
(in module operator)
(sequence method)
(str method)
(tkinter.ttk.Notebook method)
(tkinter.ttk.Treeview method)
IndexError
indexOf() (in module operator)
IndexSizeErr
is_snan() (decimal.Context method)
(decimal.Decimal method)
is_socket() (pathlib.Path method)
is_subnormal() (decimal.Decimal method)
is_symmetric() (decimal.Decimal method)
is_symlink() (os.DirEntry method)
is_tarfile() (in module tarfile)
is_term_resized() (in module curses)
is_tracing() (in module tracemalloc)
is_tracked() (in module gc)
is_unspecified
(ipaddress.IPv4Address attribute)
(ipaddress.IPv4Network attribute)
(ipaddress.IPv6Address attribute)
(ipaddress.IPv6Network attribute)
is_wintouched() (curses.window method)
is_zero() (decimal.Decimal method)
is_zipfile() (in module zipfile)
isabs() (in module os.path)
isabstract() (in module inspect)
IsADirectoryError
isalnum() (bytearray method)
(bytarray method)
(bytes method)
in module curses.ascii)
(isalpha() (bytearray method)
(bytarray method)
(bytes method)
in module curses.ascii)
isascii() (bytearray method)
(bytarray method)
(bytes method)
in module curses.ascii)
indices() (slice method)
inet_aton() (in module socket)
inet_ntoa() (in module socket)
inet_ntop() (in module socket)
inet_pton() (in module socket)
Inexact (class in decimal)
inf (in module cmath)
        (in module math)
infile
        json.tool command line option
infile (shlex.shlex attribute)
Infinity
infj (in module cmath)
info() (dis.Bytecode method)
        (gettext.NullTranslations method)
        (in module logging)
        (logging.Logger method)
infolist() (zipfile.ZipFile method)
inheritance
ini file
init() (in module mimetypes)
init_color() (in module curses)
init_database() (in module msilib)
init_pair() (in module curses)
inited (in module mimetypes)
initgroups() (in module os)
initial_indent (textwrap.TextWrapper attribute)
initialize_options()
        (distutils.cmd.Command method)
initscr() (in module curses)
inode() (os.DirEntry method)
INPLACE_ADD (opcode)
        (str method)
isasyncgen() (in module
isasyncgenfunction() (in
isatty() (chunk.Chunk meth
        (in module os)
        (io.IOBase method)
isawaitable() (in module
isblank() (in module curs
isblk() (tarfile.TarInfo me
isbuiltin() (in module insp
ischr() (tarfile.TarInfo me
isclass() (in module insp
isclose() (in module cmath
        (in module math)
iscntrl() (in module curs
iscode() (in module insp
iscoroutine() (in module
        (in module inspect)
iscoroutinefunction() (in
        (in module inspect)
isctrl() (in module curs
isDaemon() (threading.T
isdatadescriptor() (in m
isdecimal() (str method)
isdev() (tarfile.TarInfo me
isdigit() (bytearray meth
        (bytes method)
isdir() (in module os.pat
        (tarfile.TarInfo meth
isdisjoint() (frozenset me
isdown() (in module turtl
iselement() (in module x
INPLACE_AND (opcode)
INPLACE_FLOOR_DIVIDE (opcode)
INPLACE_LSHIFT (opcode)
INPLACE_MATRIX_MULTIPLY (opcode)
INPLACE_MODULO (opcode)
INPLACE_MULTIPLY (opcode)
INPLACE_OR (opcode)
INPLACE_POWER (opcode)
INPLACE_RSHIFT (opcode)
INPLACE_SUBTRACT (opcode)
INPLACE_TRUE_DIVIDE (opcode)
INPLACE_XOR (opcode)

input
(2to3 fixer)
input() (built-in function)
(in module fileinput)
input_charset (email.charset.Charset attribute)
input_codec (email.charset.Charset attribute)
InputOnly (class in tkinter.tix)
InputSource (class in xml.sax.xmlreader)
inquiry (C type)
insch() (curses.window method)
insdelln() (curses.window method)
insert() (array.array method)
(collections.deque method)
(sequence method)
(tkinter.ttk.Notebook method)
(tkinter.ttk.Treeview method)
(xml.etree.ElementTree.Element

isenabled() (in module gc)
isEnabledFor() (logging.
isendwin() (in module curses.isEOF() (in module token)
isexpr() (in module parser.ST method)
(isfifo() (tarfile.TarInfo method)
isfile() (in module os.path)
(tarfile.TarInfo method)
isfinite() (in module cmath)
(isfirstline() (in module file
isframe() (in module inspect)
isfunction() (in module in
isfuture() (in module asyncio
isgenerator() (in module async
isgeneratorfunction() (in module async
isgetsetdescriptor() (in module async
isgraph() (in module curses.ascii
isidentifier() (str method
isinf() (in module cmath)
(isinstance (2to3 fixer)
isinstance() (built-in function
iskeyword() (in module keyword)
isLeap() (in module calendar
islice() (in module itertools
islink() (in module os.path
islnk() (tarfile.TarInfo method
islower() (bytestring method
(isinstance (in module curses.a
(ismemberdescriptor() (in module

method
insert_text() (in module readline)
insertBefore() (xml.dom.Node method)
insertln() (curses.window method)
insnstr() (curses.window method)
insort() (in module bisect)
insort_left() (in module bisect)
insort_right() (in module bisect)
inspect (module)
inspect command line option
    --details
InspectLoader (class in importlib.abc)
insstr() (curses.window method)
install() (gettext.NullTranslations method)
    (in module gettext)
install_opener() (in module urllib.request)
install_scripts() (venv.EnvBuilder method)
installHandler() (in module unittest)
instance
call, [1]
class
    object, [1], [2]
instancemethod
    object
istate() (tkinter.ttk.Widget method)
instr() (curses.window method)
instream (shlex.shlex attribute)
Instruction (class in dis)
Instruction.arg (in module dis)
ismeta() (in module curses.ascii)
ismethod() (in module inspect)
ismethoddescriptor() (in module inspect)
ismodule() (in module inspect)
ismount() (in module os)
isnan() (in module cmath)
    (in module math)
ISNONTERMINAL() (in module token)
isnumeric() (str method)
isocalendar() (datetime.date method)
    (datetime.datetime method)
isoformat() (datetime.date method)
    (datetime.datetime method)
    (datetime.time method)
isolation_level (sqlite3.Connection method)
isoweekday() (datetime.date method)
    (datetime.datetime method)
isprint() (in module curses.ascii)
isprintable() (str method)
ispunct() (in module curses.ascii)
isreadable() (in module pprint)
    (pprint.PrettyPrinter method)
isrecursive() (in module pprint)
    (pprint.PrettyPrinter method)
isreg() (tarfile.TarInfo method)
isReservedKey() (http.cookies.Morsel method)
isroutine() (in module inspect)
isSameNode() (xml.dom.Node method)
isspace() (bytearray method)
    (in module curses.ascii)
    (bytes method)
    (str method)
isstdin() (in module fileinput)
issubclass() (built-in function)
Instruction.argrepr (in module dis)
Instruction.argval (in module dis)
Instruction.is_jump_target (in module dis)
Instruction.offset (in module dis)
Instruction.opcode (in module dis)
Instruction.opname (in module dis)
Instruction.starts_line (in module dis)
int
    built-in function, [1], [2]
int (built-in class)
    (uuid.UUID attribute)
Int2AP() (in module imaplib)
int_info (in module sys)
integer
    literals
    object, [1], [2]
    representation
    types, operations on
integer literal
Integral (class in numbers)
Integrated Development Environment
IntegrityError
Intel/DVI ADPCM
IntEnum (class in enum)
interact (pdb command)
interact() (code.InteractiveConsole method)
    (in module code)
    (telnetlib.Telnet method)
interactive
interactive mode
InteractiveConsole (class in code)
issubset() (frozenset method)
issuite() (in module parser (parser.ST method)
issuperset() (frozenset method)
issym() (tarfile.TarInfo method)
ISTERMINAL() (in module
istitle() (bytearray method)
    (bytes method)
    (str method)
istraceback() (in module
isub() (in module operator)
isupper() (bytearray method
    (bytes method)
    (in module curses.ascii
    (str method)
isvisible() (in moduleisxdigit() (in module curses.ascii
    (str method)
item
    sequence
    string
    item selection
item() (tkinter.ttk.Treeview
    (xml.dom.NamedNodeMap
    (xml.dom.NodeList
    itemgetter() (in module
tems() (configparser.ConfigParser
    (contextvars.Context
    (dict method)
    (email.message.Envelope
    (email.message.Message
    (mailbox.Mailbox
( types.MappingProxyType
    (xml.etree.Element
    itemsize (array.array

InteractiveInterpreter (class in code)
intern (2to3 fixer)
intern() (in module sys)
internal type
internal_attr (zipfile.ZipInfo attribute)
Internaldate2tuple() (in module imaplib)
internalSubset (xml.dom.DocumentType attribute)
Internet
interpolated string literal
interpolation
   bytestring (\%)
   bytestring (\%)
interpolation, string (\%)
InterpolationDepthError
InterpolationError
InterpolationMissingOptionError
interpolation
InterpolationSyntaxError
interpreted
interpreter
interpreter lock
interpreter prompts
interpreter shutdown
interpreter_requires_environment() (in module test.support.script_helper)
interrupt() (sqlite3.Connection method)
interrupt_main() (in module _thread)
InterruptedError
intersection() (frozenset method)
ItemsView (class in collections.abc)
ItemsView (class in typing)
iter() (built-in function)
   (xml.etree.Element method)
iter() (xml.etree.Element method)
iter_attachments() (email.message.EmailMessage method)
iter_child_nodes() (in module ast)
iter_child_nodes() (in module ast)
iter_fields() (in module ast)
iter importers() (in module pkgutil)
iter_importers() (in module pkgutil)
iter importers() (in module pkgutil)
iter_modules() (in module pkgutil)
niter modules() (in module pkgutil)
iter_parts() (email.message.EmailMessage method)
niter_unpack() (in module struct.Struct method)
iterable
   unpacking
Interable (class in collections.abc)
   (class in typing)
iterator
Interpreter (class in collections.abc)
   (class in typing)
iterator protocol
iterdecode() (in module codecs)
iterdir() (pathlib.Path method)
iterdump() (sqlite3.Connection method)
iterencode() (in module codecs)
   (json.JSONEncoder method)
iterfind() (xml.etree.Element method)
   (xml.etree.Element method)
iteritems() (mailbox.Mailbox method)
iterkeys() (mailbox.Mailbox method)
itermonthdates() (calendar method)
itermonthdays() (calendar method)
itermonthdays2() (calendar method)
itermonthdays3() (calendar method)
itermonthdays4() (calendar method)
iterparse() (in module xml.etree.ElementTree)
itertext() (xml.etree.ElementTree.Element method)

itertools (2to3 fixer) (module)
itertools_imports (2to3 fixer) (module)

itervalues() (mailbox.Mailbox method)
iterweekdays() (calendar method)

ITIMER_PROF (in module signal)
ITIMER_REAL (in module signal)
ITIMER_VIRTUAL (in module signal)

ItimerError

itruediv() (in module operator)
ixor() (in module operator)
Index – J

j

   in numeric literal

Jansen, Jack

Java
   language

java_ver() (in module platform)
join() (asyncio.Queue method)
   (bytearray method)
   (bytes method)
   (in module os.path)
   (multiprocessing.JoinableQueue method)
   (multiprocessing.Process method)
   (multiprocessing.pool.Pool method)
   (queue.Queue method)
   (str method)
   (threading.Thread method)
join_thread() (in module test.support)
   (multiprocessing.Queue method)

JoinableQueue (class in multiprocessing)
joinpath() (pathlib.PurePath method)

json
   module

json.tool (module)

json.tool command
   option
      --sort-keys
      -h, --help
      infile
      outfile

JSONDecodeError

JSONDecoder (class in json)

JSONEncoder (class in json)

jump (pdb command)

JUMP_ABSOLUTE (opcode)

JUMP_FORWARD (opcode)

JUMP_IF_FALSE_OR_POP (opcode)

JUMP_IF_TRUE_OR_POP (opcode)

js_output() (http.cookies.BaseCookie method)
   (http.cookies.Morsel method)
Index – K

kbhit() (in module msvcrt)
KDEDIR
kevent() (in module select)
key
  (http.cookies.Morsel attribute)
key function
key/datum pair
KEY_ALL_ACCESS (in module winreg)
KEY_CREATE_LINK (in module winreg)
KEY_CREATE_SUB_KEY (in module winreg)
KEY_ENUMERATE_SUB_KEYS (in module winreg)
KEY_EXECUTE (in module winreg)
KEY_NOTIFY (in module winreg)
KEY_QUERY_VALUE (in module winreg)
KEY_READ (in module winreg)
KEY_SET_VALUE (in module winreg)
KEY_WOW64_32KEY (in module winreg)
KEY_WOW64_64KEY (in module winreg)
KEY_WRITE (in module winreg)
KeyboardInterrupt

keyname() (in module curses)
keypad() (curses.window method)
keyrefs() (weakref.WeakKeyDictionary method)
keys() (contextvars.Context method)
  (dict method)
  (email.message.EmailMessage method)
  (email.message.Message method)
  (mailbox.Mailbox method)
  (sqlite3.Row method)
  (types.MappingProxyType method)
  (xml.etree.ElementTree method)
KeysView (class in collections.abc)
  (class in typing)
keyword
  as, [1], [2]
  async
  await, [1]
  elif
  else, [1], [2], [3], [4], [5]
  except
  finally, [1], [2], [3], [4]
  from, [1]
  in
  yield
keyword (module)
(built-in exception), [1]

```

(KeyError)

```

**keyword argument**

- `keywords` (functools.partial `*args`)
- `kill()`
- `(asyncio.asncio.subprocess: `method)
  - `(asyncio.SubprocessTransport: `method)
  - (in module `os`)
  - `(multiprocessing.Process: `method)
  - `(subprocess.Popen: `method)
- `kill_python()` (in `test.support.script_helper`)
- `killchar()` (in module `curses`)
- `killpg()` (in module `os`)
- `kind` (inspect.Parameter `attribute`)
- `knownfiles` (in module `mimetypes`)
- `kqueue()` (in module `select`)
- `KqueueSelector` (class in `selectors`)
- `kwargs` (inspect.BoundArguments `attribute`)
- `kwlist` (in module `keyword`)

---

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Mar 25, 2019. Found a bug?
Created using Sphinx 1.8.5.
LC_ALL, [1]  
  (in module locale)  

LC_COLLATE (in module locale)  

LC_CTYPE (in module locale)  

LC_MESSAGES, [1]  
  (in module locale)  

LC_MONETARY (in module locale)  

LC_NUMERIC (in module locale)  

LC_TIME (in module locale)  

lchflags() (in module os)  

lchmod() (in module os)  
  (pathlib.Path method)  

lchown() (in module os)  

LDCXXSHARED  

ldexp() (in module math)  

LD_FLAGS  

ldgettext() (in module gettext)  

ldngettext() (in module gettext)  

le() (in module operator)  

leading whitespace  

leapdays() (in module calendar)  

leaveok() (curses.window method)  

left (filecmp.dircmp attribute)  

left() (in module turtle)  

left_list (filecmp.dircmp attribute)  

left_only (filecmp.dircmp attribute)  

LEFTSHIFT (in module token)  

LEFTSHIFTEQUAL (in module token)  

len  
  built-in function, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]  

len() (built-in function)  

length (xml.dom.NamedNodeMap attribute)  
  (xml.dom.NodeList attribute)  

load_default_certs()  
  (ssl.SSLContext method)  

LOAD_DEREF  

load_dh_params()  

LOAD_EXTENSION  

load_extension()  
  (sqlite3.Connection method)  

LOAD_FAST (in module token)  

LOAD_GLOBAL (opcode)  

LOAD_METHOD (opcode)  

load_default_certs()  

LOAD_NAME (opcode)  

load_extension()  

load_extension()  

load_module()  
  (importlib.abc.InspectLoader method)  

load_module()  
  (importlib.abc.Loader method)  

load_module()  
  (importlib.abc.SourceLoader method)  

load_module()  
  (importlib.machinery.ModuleSpec attribute)  

load_module()  
  (importlib.machinery.SourceFileLoader method)  

load_module()  
  (importlib.abc.SourcelessFileLoader method)  

load_module()  
  (in module imp)  

load_module()  
  (zipimport.zipimporter method)  

load_package_tests()  
  (in module test.support)  

load_verify_locations()  
  (ssl.SSLContext method)  

Loader (class in importlib.abc)  

Loader (class in importlib.abc)  
  (importlib.machinery.ModuleSpec attribute)  

Loader (class in importlib.abc)  
  (importlib.machinery.Loader attribute)  

LoadError  

LoadError  

LoadKey() (in module winreg)  

LoadLibrary() (ctypes.LibraryLoader method)  

loads() (in module json)  

loads() (in module marshal)  

loads() (in module pickle)
length_hint() (in module operator)

LESS (in module token)

LESSEQUAL (in module token)

lexical analysis

lexical definitions

lexists() (in module os.path)

lgamma() (in module math)

lgettext() (gettext.GNUTranslations method)

    (gettext.NullTranslations method)

    (in module gettext)

lib2to3 (module)

libc_ver() (in module platform)

library (in module dbm.ndbm)

    (ssl.SSLLError attribute)

library_dir_option()

    (distutils.ccompiler.CCompiler method)

library_filename()

    (distutils.ccompiler.CCompiler method)

library_option() (distutils.ccompiler.CCompiler method)

LibraryLoader (class in ctypes)

license (built-in variable)

LifoQueue (class in asyncio)

    (class in queue)

light-weight processes

limit_denominator() (fractions.Fraction method)

LimitOverrunError

lin2adpcm() (in module audioop)

lin2alaw() (in module audioop)

lin2lin() (in module audioop)

lin2ulaw() (in module audioop)

line (C function)

line continuation

loadTestsFromModule()

loadTestsFromName()

loadTestsFromTestCase()

local (class in threading)

localcontext() (in module decimal)

LOCALE (in module re)

locale (module)

localeconv() (locale module)

localName (xml.dom.Node attribute)

locals() (built-in function)

Lock (class in asyncio)

    (class in multiprocessing)

    (class in threading)

lock() (mailbox.Mbox)

Lock() (multiprocessing.managers.SyncManager method)
line joining, [1]
line structure
line() (msilib.Dialog method)
line-buffered I/O
line_buffering (io.TextIOWrapper attribute)
line_num (csv.csvreader attribute)
linecache (module)
lineno (ast.AST attribute)
    (doctest.DocTest attribute)
    (doctest.Example attribute)
    (json.JSONDecodeError attribute)
    (pyclbr.Class attribute)
    (pyclbr.Function attribute)
    (re.error attribute)
    (shlex.shlex attribute)
    (traceback.TracebackException attribute)
    (tracemalloc.Filter attribute)
    (tracemalloc.Frame attribute)
    (xml.parsers.expat.ExpatError attribute)
lineno() (in module fileinput)
LINES, [1], [2], [3], [4]
lines (os.terminal_size attribute)
linesep (email.policy.Policy attribute)
    (in module os)
lineterminator (csv.Dialect attribute)
LineTooLong
link() (distutils.ccompiler.CCompiler method)
    (in module os)
link_executable()
    (distutils.ccompiler.CCompiler method)
link_shared_lib()
    (distutils.ccompiler.CCompiler method)
link_shared_object()
    (distutils.ccompiler.CCompiler method)
linkname (tarfile.TarInfo attribute)
linux_distribution() (in module platform)
**list**
  assignment, target
  comprehensions
  deletion target
  display
  empty
  expression, [1], [2]
  object, [1], [2], [3], [4], [5], [6], [7], [8]
  target, [1]
  type, operations on
list (built-in class)
List (class in typing)
list (pdb command)
**list comprehension**
list() (imaplib.IMAP4 method)
  (multiprocessing.managers.SyncManager method)
  (nntplib.NNTP method)
  (poplib.POP3 method)
  (tarfile.TarFile method)
LIST_APPEND (opcode)
list_dialects() (in module csv)
list_folders() (mailbox.Maildir method)
  (mailbox.MH method)
listdir() (in module os)
listen() (asyncore.dispatcher method)
  (in module logging.config)
  (in module turtle)
  (socket.socket method)
Listener (class in multiprocessing.connection)
listMethods()
(xmlrpc.client.ServerProxy.system method)
ListNoteBook (class in tkinter.tix)
listxattr() (in module os)
literal, [1]
literal_eval() (in module ast)
literals
  binary
  complex number
  floating point
  hexadecimal
  integer
  numeric
  octal
longname() (in module
tkinter.ttk.
symtable.
tkinter.ttk.
lookup_error()
LookupError
loop
  over mutable
  statement,
loop control
  target
loop() (in module
lower() (bytearray
  (bytes method
  (str method
LPAR (in module
lpAttributeList
attribute)
Iru_cache() (in
Iseek() (in module
Ishift() (in module
LSQB (in module
Istat() (in module
  (pathlib.Path
Istrip() (bytearray
  (bytes method
  (str method
Isub() (imaplib.
It() (in module
  (in module
LWPCookieJar
Izma (module)
LZMACompress
Index – M

M (in module re)  METH_CLASS (built-in)
mac_ver() (in module platform)  METH_COEXIST
machine() (in module platform)  METH_KEYWORDS
macpath (module)  METH_NOARGS
macros (netrc.netrc attribute)  METH_O (built-in)
MAGIC_NUMBER (in module importlib.util)  METH_STATIC (built-in)
MagicMock (class in unittest.mock)  METH_VARARGS
Mailbox (class in mailbox)  method
mailbox (module)  built-in
mailcap (module)  call
Maildir (class in mailbox)  object, [1], [2],
MaildirMessage (class in mailbox)  user-defined
mailfrom (smtpd.SMTPChannel attribute)  method (urllib.request)
MailmanProxy (class in smtplib)  method resolution
main(), [1], [2]  METHOD_BLOWFISH
    (in module py_compile)  METHOD_MD5
    (in module site)  METHOD_SHA256
    (in module unittest)  METHOD_SHA512
main_thread() (in module threading)  methodattrs (2to3 fixer)
mainloop() (in module turtle)  methodcaller()
maintype  (in module 3to2 fixer)
(email.headerregistry.ContentTypeHeader attribute)  MethodDescriptorType
major  (in module types)
(email.headerregistry.MIMEVersionHeader attribute)  methodHelp()
major() (in module os)  (xmlrpc.client.ServerProxy)
make_alternative()  method
(email.message.EmailMessage method)  methods
make_archive()  bytearray
    (in module string)  bytes
    string
makeelement()
(xml.etree.ElementTree.Element method)
makefile()
(socket method)
(makefile.socket method)
makeLogRecord()
(in module logging)
makePickle()
(logging.handlers.SocketHandler method)
makeRecord()
(logging.Logger method)
makeSocket()
(logging.handlers.DatagramHandler method)
(making_socket.socket method)
(making_socket.socket method)
maketrans()
(bytestring static method)
(bytestring static method)
(bytestring static method)
malloc()

mangle_from_
(email.policyCompat32 attribute)
(email.policy.Policy attribute)
mangling

name, [1]
map (2to3 fixer)
map()
(built-in function)
(map.stringprep.Json method)
(map.stringprep.Json method)
(map.stringprep.Json method)
(map.stringprep.Json method)
(map.stringprep.Json method)
(MAP_ADD opcode)
map_async()
(multiprocessing.pool.Pool method)
map_table_b2()
(in module stringprep)
minor()
(multiprocessing.pool.Pool method)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)
MIN_YEAR
(min_year 
minus)
MINUS (in module
time)
minute
datetime.datetime attribute)}
method)
mmapLogRecord()
(logging.handlers.HTTPHandler method)
**mapping**
  object, [1], [2], [3], [4], [5]
  types, operations on
Mapping (class in collections.abc)
  (class in typing)
mapping() (msilib.Control method)
MappingProxyType (class in types)
MapView (class in collections.abc)
  (class in typing)
mapPriority()
(logging.handlers.SysLogHandler method)
maps (collections.ChainMap attribute)
maps() (in module nis)
marshal (module)
marshalling
  objects
masking
  operations
Match (class in typing)
mach() (in module nis)
  (in module re)
  (pathlib.PurePath method)
  (re.Pattern method)
m_match_hostname() (in module ssl)
m_match_test() (in module test.support)
m_match_value() (test.support.Matcher method)
Matcher (class in test.support)
m_matches() (test.support.Matcher method)
math
  module, [1]
  test.support
  missing_compiler_executable()
  MissingSectionHeaderError
  MIXERDEV
mkd() (ftplib.FTP method)
mkdir() (in module os)
  (pathlib.Path method)
mkdtemp() (in module tempfile)
mkfifo() (in module os)
mknod() (in module os)
mkpath() (distutils.ccompiler.CCompiler method)
  (in module distutils.dir_util)
mksalt() (in module crypt)
mkstemp() (in module tempfile)
mktemp() (in module tempfile)
mktemp() (in module time)
mktime_tz() (in module email.utils)
mlsd() (ftplib.FTP method)
mmap (class in mmap module)
MMDF (class in mailbox)
MMDFMessage (class in mailbox)
Mock (class in unittest)
mock_add_spec() (unittest.mock method)
mock_calls (unittest.mock method)
mock_open() (in modules)
mkmod() (in module os)
mode (io.FileIO attribute)
  (ossaudiodev.attribute)
  (tarfile.TarInfo attribute)
math (module)
matmul() (in module operator)
matrix multiplication
max
    built-in function
max (datetime.date attribute)
    (datetime.datetime attribute)
    (datetime.timedelta attribute)
max() (built-in function)
    (decimal.Context method)
    (decimal.Decimal method)
    (in module audioop)
max_count
(email.headerregistry.BaseHeader attribute)
MAX_EMAX (in module decimal)
MAX_INTERPOLATION_DEPTH (in module configparser)
max_line_length (email.policy.Policy attribute)
max_lines (textwrap.TextWrapper attribute)
max_mag() (decimal.Context method)
    (decimal.Decimal method)
max_memuse (in module test.support)
MAX_PREC (in module decimal)
max_prefixlen (ipaddress.IPv4Address attribute)
    (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Address attribute)
    (ipaddress.IPv6Network attribute)
MAX_Py_ssize_t (in module test.support)
maxarray (reprlib.Repr attribute)
mode() (in module
turtle)
  (in module)
modes
  file
  modf() (in module
  modified() (urllib.robotparser.
    method)
  Modify() (msilib.View method)
  modify() (select.de
    (select.epoll
    (select.poll
    (selectors.BaseSelector
module
    __main__, [1]
_locale
_thread
array, [1]
base64
bdb
binhex
builtins, [1], [2
cmd
copy
crypt
dbm.gnu, [1]
dbm.ndbm, [1
errno
extension
glob
imp
importing
io
json
median_high() (in module statistics)
median_low() (in module statistics)
MemberDescriptorType (in module types)
membership
test
memmove() (in module ctypes)
MemoryBIO (class in ssl)
MemoryError
MemoryHandler (class in logging.handlers)
memoryview
    object, [1]
memoryview (built-in class)
memset() (in module ctypes)
merge() (in module heapq)
Message (class in email.message)
    (class in mailbox)
message digest, MD5
message_factory (email.policy.Policy attribute)
message_from_binary_file() (in module email)
message_from_bytes() (in module email)
message_from_file() (in module email)
message_from_string() (in module email)
MessageBeep() (in module winsound)
MessageClass (http.server.BaseHTTPRequestHandler attribute)
ErrorMessage
MessageParseError
messages (in module xml.parsers.expat.errors)
meta

ModuleInfo (class in modulefinder)
ModuleNotFoundError
modules (in module sys)
    (modulefinder.ModuleFinder attribute)
modules_cleanup() (in module test.support)
modules_setup() (in module test.support)
ModuleSpec (class in importlib.machinery)
ModuleType (class in module types)
modulo
monotonic() (in module monotonic)
monotonic_ns() (in module monotonic)
month (datetime.date attribute)
    (datetime.datetime attribute)
month() (in module calendar)
month_abbr (in module calendar)
month_name (in module calendar)
monthcalendar() (in module calendar)
monthdays2calendar() (in module calendar)
monthdayscalendar() (in module calendar)
monthrange() (in module calendar)
Morsel (class in http.cookies)
most_common() (in module collections)
mouseinterval() (in module curses)
mousemask() (in module curses)
mov() (curses.panel.Panel method)
    (curses.window method)
move() (in module shutil)
hooks
meta hooks

**meta path finder**

meta() (in module curses)
meta_path (in module sys)
metaclass, [1]  
(formerly 2to3 fixer)
metaclass hint
MetaPathFinder (class in importlib.abc)
metavar (optparse.Option attribute)
MetavarTypeHelpFormatter (class in argparse)
Meter (class in tkinter.tix)

(mmap.mmap method)
(tkinter.ttk.Treeview method)
move_file() (distutils.ccompiler.CCompiler method)
move_to_end() (collections.OrderedDict method)
MozillaCookieJar (module)
mro() (class method)
msg (http.client.HTTPResponse attribute)
(msg() (re.error attribute)
(msg() (traceback.TracebackException attribute)
msg() (telnetlib.Telnet method)
msi (module)
msilib (module)
msvcrt (module)
mt_interact() (telnetlib.Telnet method)
mtime (gzip.GzipFile attribute)
mtime () (tarfile.TarInfo method)
msg() (urllib.robotparser.RobotFileParser method)
mul() (in module audioop)
multiprocessing (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
multiprocessing.connection (module)
mutable

object, [1], [2]

sequence types

mutable object

mutable sequence

loop over

object

MutableMapping (class in typing)

MutableSequence (class in collections.abc)

MutableSet (class in typing)

mvderwin() (curses.window method)

mvwin() (curses.window method)

myrights() (imaplib.IMAP4 method)
Index – N

N_TOKENS (in module token)  NNTPDataErr
n_waiting (threading.Barrier attribute)  NNTPError
name, [1], [2]  nntplib (module)
  binding, [1], [2], [3], [4], [5], [6]  NTTPPermanent
  binding, global  NTTPProtocol
  class  NTTPReplyError
  function, [1]  NOEXPR (in module locale)
  mangling, [1]  NoModificationAllowedErr
  rebinding  nonblock()
  unbinding  non_type_check
name (codecs.CodecInfo attribute)  non_type_check_typing
  (contextvars.ContextVar attribute)  no_proxy
  (doctest.DocTest attribute)  no_tracing()
  (email.headerregistry.BaseHeader module typing
   attribute)  test.support)
  (hashlib.hash attribute)  nocbreak() (in
  (hmac.HMAC attribute)  NoDataAllowedErr
  (http.cookiejar.Cookie attribute)  node() (in module
  (importlib.abc.FileLoader attribute)  nodelay()
  (importlib.machinery.ExtensionFileLoader method)
   attribute)  noecho()
  (importlib.machinery.ModuleSpec nodeName
   attribute)  NodeTransformer
  attribute)  nodeType
  (importlib.machinery.SourceFileLoader  nodeValue
   attribute)  NodeVisitor (c
  (importlib.machinery.SourcelessFileLoader  noecho() (in
   attribute)  no_proxy
  (importlib.machinery.SourcelessFileLoader  NOEXPR (in token
   attribute)  nonblock())
(multiprocessing.Process attribute)
(os.DirEntry attribute)
(ossaudiodev.oss_audio_device attribute)
(pyclbr.Class attribute)
(pyclbr.Function attribute)
(tarfile.TarInfo attribute)
(threading.Thread attribute)
(xml.dom.Attr attribute)
(xml.dom.Document attribute)

name() (in module unicodedata)
name2codepoint (in module html.entities)

**named tuple**
NamedTemporaryFile() (in module tempfile)
NamedTuple (class in typing)
namedtuple() (in module collections)

NameError

NonCallableMock (class in unittest.mock)
None
NonCallableMock in unittest.mock

None (Built-in object)
nonl() (in module curses)
nonlocal

normalvariate() (in module random)
normcase() (in module os.path)

namespace, [1]

namespace package
namespace() (imaplib.IMAP4 method)
Namespace()
(multiprocessing.managers.SyncManager method)

NAMESPACE_DNS (in module uuid)
NAMESPACE_OID (in module uuid)
NAMESPACE_URL (in module uuid)
NAMESPACE_X500 (in module uuid)
NamespaceErr
namespaceURI (xml.dom.Node attribute)
NaN
nan (in module cmath)
    (in module math)
nanj (in module cmath)
NannyNag
napms() (in module curses)
nargs (optparse.Option attribute)
nbytes (memoryview attribute)
ndiff() (in module difflib)
ndim (memoryview attribute)
ne (2to3 fixer)
ne() (in module operator)
needs_input (bz2.BZ2Decompressor attribute)
    (lzma.LZMADecompressor attribute)
neg() (in module operator)
negation

**nested scope**
netmask (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Network attribute)
NetmaskValueError
netrc (class in netrc)
    (module)
NetrcParseError
netscape (http.cookiejar.CookiePolicy attribute)
network (ipaddress.IPv4Interface attribute)
    (ipaddress.IPv6Interface attribute)
normpath() (in os.path)

NoSectionError
NoSuchMailboxError
not
operator,
not() (in module operator)

NotADirectoryError
notation
notationDecl()
    (xml.sax.handler.DTDHandler method)
notations
(xml.dom.Document attribute)
NotConnected
NoteBook (class in tkinter.tix)
Notebook (class in tkinter.ttk)
NotEmptyError
NOTEQUAL (in module token)
NotFoundErr
notify()
    (asyncio.Condition method)
    (threading.Condition method)
notify_all()
    (asyncio.Condition method)
    (threading.Condition method)
notimeout()
Network News Transfer Protocol

network_address  (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Network attribute)
new()  (in module hashlib)
    (in module hmac)
**new-style class**
new_alignment()  (formatter.writer method)
new_child()  (collections.ChainMap method)
new_class()  (in module types)
new_compiler()  (in module distutils.ccompiler)
new_event_loop()
    (asyncio(AbstractEventLoopPolicy method)
        (in module asyncio)
new_font()  (formatter.writer method)
new_margin()  (formatter.writer method)
new_module()  (in module imp)
new_panel()  (in module curses.panel)
new_spacing()  (formatter.writer method)
new_styles()  (formatter.writer method)
newer()  (in module distutils.dep_util)
newer_group()  (in module distutils.dep_util)
newer_pairwise()  (in module distutils.dep_util)
newgroups()  (nntplib.NNTP method)
NEWLINE  (in module token)
NEWLINE token, [1]
newlines  (io.TextIOBase attribute)
newnews()  (nntplib.NNTP method)
newpad()  (in module curses)
NewType()  (in module typing)
newwin()  (in module curses)
next  (2to3 fixer)
    (pdb command)
next()  (built-in function)
(nntplib.NNTP method) NullWriter (class in formatter)
(tarfile.TarFile method) num_address
(tkinter.ttk.Treeview method) (ipaddress.IPv4Network attribute)
next_minus() (decimal.Context method) (ipaddress.IPv6Network attribute)
    (decimal.Decimal method)
next_plus() (decimal.Context method) number
    (decimal.Decimal method)
next_toward() (decimal.Context method) complex
    (decimal.Decimal method)
nextfile() (in module fileinput) floating point
nextkey() (dbm.gnu.gdbm method) Number (class in numbers)
nexSibling (xml.dom.Node attribute) NUMBER (in module numbers)
ngettext() (gettext.GNUTranslations method) number_class
    (gettext.NullTranslations method)
    (decimal.Decimal method)
nlargest() (in module heapq) numbers (module numbers)
nlst() (ftplib.FTP method)
nl() (in module curses)
nl_langinfo() (in module locale)
nlargest() (in module heapq)
numeric
numeric
numeric() convrsic
literals
object, [1:
numeric literal
types, op
unicodedata)
unumeric
numinput() (in module numerals)
unuminput() (in
nuliters (2)}
Index – O

O_APPEND (in module os)
O_ASYNC (in module os)
O_BINARY (in module os)
O_CLOEXEC (in module os)
O_CREAT (in module os)
O_DIRECT (in module os)
O_DIRECTORY (in module os)
O_DSYNC (in module os)
O_EXCL (in module os)
O_EXLOCK (in module os)
O_NDELAY (in module os)
O_NOATIME (in module os)
O_NOCTTY (in module os)
O_NOFOLLOW (in module os)
O_NOINHERIT (in module os)
O_NONBLOCK (in module os)
O_PATH (in module os)
O_RANDOM (in module os)
O_RDONLY (in module os)
O_RDWR (in module os)
O_RSYNC (in module os)
O_SEQUENTIAL (in module os)
O_SHLOCK (in module os)
O_SHORT_LIVED (in module os)
O_SYNC (in module os)
O_TEMPORARY (in module os)
O_TEXT (in module os)
O_TMPFILE (in module os)
O_TRUNC (in module os)
O_WRONLY (in module os)

obj (memoryview attribute)

open_osfhandle() (in module msvcrt)
open_resource() (in module importlib.abc.ResourceReader method)
open_text() (in module importlib.resources)
open_unix_connection (in module asyncio)
open_unknown() (in module urllib.request.URLopener method)
open_urlresource() (in module test.support)
OpenDatabase() (in module msilib)
OpenerDirector (in module urllib.request)
openfp() (in module s in module wave)
OpenKey() (in module winreg)
OpenKeyEx() (in module winreg)
openlog() (in module syslog)
opensm() (in module ossaudiodev)
openssl() (in module OpenSSL)
(use in module hashlib)
(use in module ssl)
OPENSSL_VERSION (in module ssl)
OPENSSL_VERSION_INFO (in module ssl)
OPENSSL_VERSION_NUMBER (in module ssl)
object, [1]
  Boolean, [1]
  Capsule
  Ellipsis
  None, [1], [2]
  Not Implemented
  asynchronous-generator
  built-in function, [1]
  built-in method, [1]
  bytearray, [1], [2], [3]
  bytes, [1], [2]
  callable, [1]
  class, [1], [2]
  class instance, [1], [2]
  code, [1], [2], [3]
  complex
  complex number, [1]
  deallocation
  dictionary, [1], [2], [3], [4], [5], [6], [7]
  file, [1]
  finalization
  floating point, [1], [2]
  frame
  frozenset, [1]
  function, [1], [2], [3], [4], [5]
  generator, [1], [2]
  immutable, [1], [2]
  immutable sequence
  instance, [1], [2]
  instancemethod
  integer, [1], [2]
  io.StringIO
  list, [1], [2], [3], [4], [5], [6], [7], [8]

OpenView() (msilib.Database	method)
operation
  Boolean
  binary arithmetic
  binary bitwise
  concatenation
  null, [1]
  power
  repetition
  shifting
  slice
  subscript
  unary arithmetic
  unary bitwise
OperationalError
operations
  Boolean, [1]
  bitwise
  masking
  shifting
operations on
  dictionary type
  integer types
  list type
  mapping types
  numeric types
  sequence types,
operator
  !=, [1]
  % (percent), [1]
  & (ampersand),
  * (asterisk), [1]
  **, [1]
  + (plus), [1], [2]
long integer
mapping, [1], [2], [3], [4], [5]
memoryview, [1]
method, [1], [2], [3], [4], [5]
module, [1], [2]
mutable, [1], [2]
mutable sequence
numeric, [1], [2], [3], [4]
range
sequence, [1], [2], [3], [4], [5], [6], [7], [8]
set, [1], [2], [3]
set type
slice
socket
string, [1], [2]
traceback, [1], [2], [3], [4]
tuple, [1], [2], [3], [4], [5], [6]
type, [1], [2]
user-defined function, [1], [2]
user-defined method
object (built-in class)
  (UnicodeError attribute)
object.__slots__ (built-in variable)
object_filenames()
(distutils.ccompiler.CCompiler method)
objects
  comparing
  flattening
  marshalling
  persistent
  pickling
  serializing
  obufcount() - (minus), [1], [2]
  / (slash), [1]
  //, [1]
  < (less), [1]
  <<, [1]
  <=, [1]
  ==, [1]
  > (greater), [1]
  >=, [1]
  @ (at)
  ^ (caret), [1]
  and, [1], [2]
  comparison
  in, [1], [2]
  is, [1]
  is not, [1]
  not, [1]
  not in, [1], [2]
  or, [1], [2]
  overloading
  precedence
  ternary
  operator (2to3 fixer)
  (module)
  operators
  opmap (in module dis)
  opname (in module code)
  optim_args_from_interpreter_flags() (in module test.support)
  optimize() (in module compile)
  OPTIMIZED_BYTECODE_SUFFIXES
OP_NO_COMPRESSION (in module ssl)
OP_NO_RENEGOTIATION (in module ssl)
OP_NO_SSLv2 (in module ssl)
OP_NO_SSLv3 (in module ssl)
OP_NO_TICKET (in module ssl)
OP_NO_TLSv1 (in module ssl)
OP_NO_TLSv1_1 (in module ssl)
OP_NO_TLSv1_2 (in module ssl)
OP_NO_TLSv1_3 (in module ssl)
OP_SINGLE_DH_USE (in module ssl)
OP_SINGLE_ECDH_USE (in module ssl)

open (built-in function, [1])
open() (built-in function)

(distutils.text_file.TextFile method)
(imaplib.IMAP4 method)
(in module aifc)
(in module bz2)
(in module codecs)
(in module dbm)
(in module dbm.dumb)
(in module dbm.gnu)
(in module dbm.ndbm)
(in module gzip)
(in module io)
(in module lzma)
(in module os)
(in module ossaudiodev)
(in module shelf)
(in module sunau)
(in module tarfile)

(output (http.cookies.Morsel method)
(output_charset (email.charset.Charset attribute))
(output_difference())

origin_server
(wsgiref.handlers.BaseHandler attribute)

os
(module)

os (module)

os.path (module)

os_environ
(wsgiref.handlers.BaseHandler attribute)

OSError

ossaudiodev (module)

OSSAudioError

outfile

json.tool command line option

output

standard output

(subprocess.CalledProcessError attribute)

(subprocess.TimeoutExpired attribute)

(unittest.TestCase attribute)

output() (http.cookies.BaseCookie method)

(output_charset (email.charset.Charset attribute))

output_charset()

(gettext.NullTranslations method)

output_codec (email.charset.Charset attribute)

output_difference()
Index – P

p (pdb command)  PyEval_
P_ALL (in module os)  PyEval_
P_DETACH (in module os)  PyEval_
P_NOWAIT (in module os)  PyEval_
P_NOWAITO (in module os)  PyEval_
P_OVERLAY (in module os)  PyEval_
P_PGID (in module os)  PyEval_
P_PID (in module os)  PyEval_
P_WAIT (in module os)  PyEval_
pack() (in module struct)  PyEval_
   (mailbox.MH method)  PyEval_
   (struct.Struct method)  PyEval_
pack_array() (xdrlib.Packer method)  PyEval_
pack_bytes() (xdrlib.Packer method)  PyEval_
pack_double() (xdrlib.Packer method)  PyEval_
pack_farray() (xdrlib.Packer method)  PyEval_
pack_float() (xdrlib.Packer method)  PyEval_
pack_fopaque() (xdrlib.Packer method)  PyEval_
pack_fstring() (xdrlib.Packer method)  PyEval_
pack_into() (in module struct)  PyExc_
   (struct.Struct method)  PyExc_
pack_list() (xdrlib.Packer method)  PyExc_
pack_opaque() (xdrlib.Packer method)  PyExc_
pack_string() (xdrlib.Packer method)  PyExc_
package, [1], [2]  PyExc_
   namespace  PyExc_
   portion  PyExc_
   regular  PyExc_
Package (in module importlib.resources)  PyExc_
package variable  PyExc_
   __all__  PyExc_
packed (ipaddress.IPv4Address attribute)

   (ipaddress.IPv6Address attribute)

Packer (class in xdrlib)

packing

   binary data

packing (widgets)

PAGER

pair_content() (in module curses)

pair_number() (in module curses)

PanedWindow (class in tkinter.tix)

**parameter**

   call semantics

   difference from argument

   function definition

   value, default

Parameter (class in inspect)

ParameterizedMIMEHeader (class in email.headerregistry)

parameters (inspect.Signature attribute)

params

(email.headerregistry.ParameterizedMIMEHeader attribute)

pardir (in module os)

paren (2to3 fixer)

parent (importlib.machinery.ModuleSpec attribute)

   (pyclbr.Class attribute)

   (pyclbr.Function attribute)

   (urllib.request.BaseHandler attribute)

parent() (tkinter.ttk.Treeview method)

parenthesized form

parentNode (xml.dom.Node attribute)

parents (collections.ChainMap attribute)

paretovariate() (in module random)

parse() (doctest.DocTestParser method)

PyExc_ConnectionResetError

PyExc_DeprecationWarning

PyExc_EnvironmentError

PyExc_EOFError

PyExc_Exception

PyExc_FileExistsError

PyExc_FileNotFoundError

PyExc_FloatingPointError

PyExc_FutureWarning

PyExc_GeneratorExit

PyExc_ImportError

PyExc_ImportWarning

PyExc_IndentationError

PyExc_IndexError

PyExc_InterruptedError

PyExc_IOError

PyExc_IsADirectoryError

PyExc_KeyboardInterrupt

PyExc_KeyError

PyExc_LookupError

PyExc_MemoryError

PyExc_ModuleNotFoundError

PyExc_NameError

PyExc_NotADirectoryError

PyExc_NotImplementedError

PyExc_OSError

PyExc_OverflowError

PyExc_PendingDeprecationWarning

PyExc_PermissionError

PyExc_ProcessLookupError

PyExc_RecursionError

PyExc_ReferenceError

PyExc_ResourceWarning

PyExc_RuntimeError
parsedate_tz() (in module email.utils)
ParseError (class in xml.etree.ElementTree)
ParseFile() (xml.parsers.expat.xmlparser method)
ParseFlags() (in module imaplib)
parser
Parser (class in email.parser)
parserr (module)
ParserCreate() (in module xml.parsers.expat)
ParserError
ParseResult (class in urllib.parse)
ParseResultBytes (class in urllib.parse)
parsestr() (email.parser.Parser method)
parseString() (in module xml.dom.minidom)
  (in module xml.dom.pulldom)
  (in module xml.sax)
parsing
  Python source code
  URL
ParsingError
partial (asyncio.IncompleteReadError attribute)
partial() (imaplib.IMAP4 method)
  (in module functools)
partialmethod (class in functools)
parties (threading.Barrier attribute)
partition() (bytearray method)
  (bytes method)
  (str method)
pass
  statement
pass_() (poplib.POP3 method)
Paste
patch() (in module test.support)
  (in module unittest.mock)
patch.dict() (in module unittest.mock)
patch.multiple() (in module unittest.mock)
patch.object() (in module unittest.mock)
patch.stopall() (in module unittest.mock)
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34]
path
(configuration file)
(hooks)
(module search, [1], [2], [3], [4], [5], [6], [7]
(operations, [1]
Path (class in pathlib)
path (http.cookiejar.Cookie attribute)
(http.server.BaseHTTPRequestHandler attribute)
(importlib.abc.FileLoader attribute)
(importlib.machinery.ExtensionFileLoader attribute)
(importlib.machinery.FileFinder attribute)
(importlib.machinery.SourceFileLoader attribute)
(importlib.machinery.SourcelessFileLoader attribute)
(in module sys), [1], [2], [3], [4]
(os.DirEntry attribute)
(path based finder, [1]
Path browser
path entry
path entry finder
path entry hook
path hooks
path() (in module importlib.resources)
path-like object
path_hook() (importlib.machinery.FileFinder class
method)  function
path_hooks (in module sys)  PyImport
path_importer_cache (in module sys)  PyImport
path_mtime() (importlib.abc.SourceLoader method)  PyIndex
path_return_ok() (http.cookiejar.CookiePolicy method)  PyInstan
path_stats() (importlib.abc.SourceLoader method)  PyInstan
    (importlib.machinery.SourceFileLoader method)  PyInstan
pathconf() (in module os)  function
pathconf_names (in module os)  PyInstan
PathEntryFinder (class in importlib.abc)  PyInstan
PATH_EXT, [1]  PyInterp
PathFinder (class in importlib.machinery)  PyInterp
pathlib (module)  PyInterp
PathLike (class in os)  PyInterp
pathname2url() (in module urllib.request)  PyInterp
pathsep (in module os)  PyInterp
Pattern (class in typing)  PyInterp
pattern (re.error attribute)  PyInterp
    (re.Pattern attribute)  PyInterp
pause() (in module signal)  PyInterp
pause_reading() (asyncio.ReadTransport method)  PyList
pause_writing() (asyncio.BaseProtocol method)  PyList
PAX_FORMAT (in module tarfile)  PyList
pax_headers (tarfile.TarFile attribute)  PyList
    (tarfile.TarInfo attribute)  PyList
pbkdf2_hmac() (in module hashlib)  PyList
pd() (in module turtle)  PyList
Pdb (class in pdb), [1]  PyList
pdb (module)  PyList
peek() (bz2.BZ2File method)  PyList
    (gzip.GzipFile method)  PyList
    (io.BufferedReader method)  PyList
    (lzma.LZMAFile method)  PyList

(weakref.finalize method)
peer (smtpd.SMTPChannel attribute)
PEM_cert_to_DER_cert() (in module ssl)
pen() (in module turtle)
pencolor() (in module turtle)
pending (ssl.MemoryBIO attribute)
pending() (ssl.SSLSocket method)
PendingDeprecationWarning
pendown() (in module turtle)
pensize() (in module turtle)
penup() (in module turtle)
PEP
PERCENT (in module token)
PERCENTEQUAL (in module token)
perf_counter() (in module time)
perf_counter_ns() (in module time)
Performance
PermissionError
permutations() (in module itertools)
Persist() (msilib.SummaryInformation method)
persistence
persistent
    objects
persistent_id (pickle protocol)
persistent_id() (pickle.Pickler method)
persistent_load (pickle protocol)
persistent_load() (pickle.Unpickler method)
PF_CAN (in module socket)
PF_PACKET (in module socket)
PF_RDS (in module socket)
pformat() (in module pprint)
    (pprint.PrettyPrinter method)
PGO (in module test.support)
phase() (in module cmath)
Philbrick, Geoff

physical line, [1], [2]

pi (in module cmath)
  (in module math)

pickle
  module, [1], [2], [3]
pickle (module)
pickle() (in module copyreg)
PickleError

Pickler (class in pickle)
pickletools (module)
pickletools command line option
  -a, --annotate
  -l, --indentlevel=<num>
  -m, --memo
  -o, --output=<file>
  -p, --preamble=<preamble>
pickling
  objects

PicklingError

pid (asyncio.asyncio.subprocess.Process attribute)
  (multiprocessing.Process attribute)
  (subprocess.Popen attribute)

PIPE (in module subprocess)
Pipe() (in module multiprocessing)
pipe() (in module os)
pipe2() (in module os)
PIPE_BUF (in module select)

pipe_connection_lost() (asyncio.SubprocessProtocol method)
pipe_data_received() (asyncio.SubprocessProtocol method)

PIPE_MAX_SIZE (in module test.support)
pipes (module)
PKG_DIRECTORY (in module imp)
npkgutil (module)
placeholder (textwrap.TextWrapper attribute)
PLAT
platform (in module sys), [1]
   (module)
platform() (in module platform)
PlaySound() (in module winsound)
plist
   file
plistlib (module)
plock() (in module os)
plus
PLUS (in module token)
plus() (decimal.Context method)
PLUSEQUAL (in module token)
pm() (in module pdb)
POINTER() (in module ctypes)
pointer() (in module ctypes)
polar() (in module ccmath)
Policy (class in email.policy)
poll() (in module select)
   (multiprocessing.connection.Connection method)
   (select.devpoll method)
   (select.epoll method)
   (select.poll method)
   (subprocess.Popen method)
PollSelector (class in selectors)
Pool (class in multiprocessing.pool)
pop() (array.array method)
   (collections.deque method)
   (dict method)
   (frozenset method)
(mailbox.Mailbox method)
(mailbox.Mailbox method)
(sequence method)

POP3 protocol

POP3 (class in poplib)

POP3_SSL (class in poplib)

pop_alignment() (formatter.formatter method)

pop_all() (contextlib.ExitStack method)

POP_BLOCK (opcode)

POP_EXCEPTION (opcode)

pop_font() (formatter.formatter method)

POP_JUMP_IF_FALSE (opcode)

POP_JUMP_IF_TRUE (opcode)

pop_margin() (formatter.formatter method)

pop_source() (shlex.shlex method)

pop_style() (formatter.formatter method)

POP_TOP (opcode)

Popen (class in subprocess)

popen() (in module os), [1], [2]

(popen() (in module platform)

popitem() (collections.OrderedDict method)

(popitem() (collections.OrderedDict method)

popleft() (collections.deque method)

poplib (module)

PopupMenu (class in tkinter.tix)

port (http.cookiejar.Cookie attribute)

port_specified (http.cookiejar.Cookie attribute)

portion package

pos (json.JSONDecodeError attribute)

(re.Match attribute)

(re.error attribute)

pos() (in module operator)
position (xml.etree.ElementTree.ParseError attribute)  (in module turtle)

position() (in module turtle)

**positional argument**

POSIX

I/O control

threads

posix (module)

POSIX_FADV_DONTNEED (in module os)

POSIX_FADV_NOREUSE (in module os)

POSIX_FADV_NORMAL (in module os)

POSIX_FADV_RANDOM (in module os)

POSIX_FADV_SEQUENTIAL (in module os)

POSIX_FADV_WILLNEED (in module os)

posix_fadvise() (in module os)

posix_fallocate() (in module os)

POSIXLY_CORRECT

PosixPath (class in pathlib)

post() (nntplib.NNTP method)

(post_handshake_auth (ssl.SSLContext attribute)

post_mortem() (in module pdb)

post_setup() (venv.EnvBuilder method)

postcmd() (cmd.Cmd method)

postloop() (cmd.Cmd method)

pow

built-in function, [1], [2], [3], [4], [5]

pow() (built-in function)

(in module math)

(in module operator)

power

operation

power() (decimal.Context method)

pp (pdb command)
pprint (module)
pprint() (in module pprint)  (pprint.PrettyPrinter method)
prcal() (in module calendar)
pread() (in module os)
preadv() (in module os)
preamble (email.message.EmailMessage attribute)  (email.message.Message attribute)
precmd() (cmd.Cmd method)
prefix, [1], [2], [3]
PREFIX (in module distutils.sysconfig)
prefix (in module sys)
	(xml.dom.Attr attribute)
	(xml.dom.Node attribute)
	(zipimport.zipimporter attribute)
PREFIXES (in module site)
prefixlen (ipaddress.IPv4Network attribute)
	(ipaddress.IPv6Network attribute)
preloop() (cmd.Cmd method)
prepare() (logging.handlers.QueueHandler method)
	(logging.handlers.QueueListener method)
prepare_class() (in module types)
prepare_input_source() (in module xml.sax.saxutils)
p pend() (pipes.Template method)
preprocess() (distutils.ccompiler.CCompiler method)
PrettyPrinter (class in pprint)
prev() (tkinter.ttk.Treeview method)
previousSibling (xml.dom.Node attribute)
primary
print  built-in function
print (2to3 fixer)
print() (built-in function)  
  _str__() (object method)

print_callees() (pstats.Stats method)  
print_callers() (pstats.Stats method)  
print_directory() (in module cgi)  
print_environ() (in module cgi)  
print_environ_usage() (in module cgi)  
print_exc() (in module traceback)  
  (timeit.Timer method)

print_exception() (in module traceback)  
PRINT_EXPR (opcode)  
print_form() (in module cgi)  
print_help() (argparse.ArgumentParser method)  
print_last() (in module traceback)  
print_stack() (asyncio.Task method)  
  (in module traceback)

print_stats() (profile.Profile method)  
  (pstats.Stats method)

print_tb() (in module traceback)  
print_usage() (argparse.ArgumentParser method)  
  (optparse.OptionParser method)

print_version() (optparse.OptionParser method)  
printable (in module string)  
printdir() (zipfile.ZipFile method)  
printf-style formatting, [1]  
PRIO_PGRP (in module os)  
PRIO_PROCESS (in module os)  
PRIO_USER (in module os)  
PriorityQueue (class in asyncio)  
  (class in queue)

private  
  names  
prlimit() (in module resource)  
prmonth() (calendar.TextCalendar method)
Profile (class in profile)
profile (module)
profile function, [1], [2]
profiler, [1]
profiling, deterministic
program
ProgrammingError
Progressbar (class in tkinter.ttk)
prompt (cmd.Cmd attribute)
prompt_user_passwd()
(urllib.request.FancyURLopener method)
prompts, interpreter
propagate (logging.Logger attribute)
property (built-in class)
property list
property_declaration_handler (in module xml.sax.handler)
property_dom_node (in module xml.sax.handler)
property_lexical_handler (in module xml.sax.handler)
property_xml_string (in module xml.sax.handler)
PropertyMock (class in unittest.mock)
prot_c() (ftplib.FTP_TLS method)
prot_p() (ftplib.FTP_TLS method)
proto (socket.socket attribute)
protocol
  CGI
  FTP, [1]
  HTTP, [1], [2], [3], [4], [5]
  IMAP4
  IMAP4_SSL
  IMAP4_stream
  NNTP
  POP3
  SMTP
PyParser_SimpleParseString
PyParser_SimpleParseStringFlags
PyParser_SimpleParseStringFlagsFilename
PyPI
PyProperty_Type
PyRun_AnyFile
PyRun_AnyFileEx
PyRun_AnyFileExFlags
PyRun_AnyFileFlags
PyRun_InteractiveLoop
PyRun_InteractiveLoopFlags
PyRun_InteractiveOne
PyRun_InteractiveOneFlags
PyRun_SimpleFile
PyRun_SimpleFileEx
PyRun_SimpleFileExFlags
PyRun_SimpleString
PyRun_SimpleStringFlags
PyRun_String
PyRun_StringFlags
PySeqIter_Check
PySeqIter_New
PySeqIter_Type
PySequence_Check
PySequence_Concat
PySequence_Contains
PySequence_Count
Telnet
context management
copy
iterator
Protocol (class in asyncio)
protocol (ssl.SSLContext attribute)
PROTOCOL_SSLv2 (in module ssl)
PROTOCOL_SSLv23 (in module ssl)
PROTOCOL_SSLv3 (in module ssl)
PROTOCOL_TLS (in module ssl)
PROTOCOL_TLS_CLIENT (in module ssl)
PROTOCOL_TLS_SERVER (in module ssl)
PROTOCOL_TLSv1 (in module ssl)
PROTOCOL_TLSv1_1 (in module ssl)
PROTOCOL_TLSv1_2 (in module ssl)
protocol_version
(http.server.BaseHTTPRequestHandler attribute)
PROTOCOL_VERSION (imaplib.IMAP4 attribute)
ProtocolError (class in xmlrpc.client)
provisional API
provisional package
proxy() (in module weakref)
proxyauth() (imaplib.IMAP4 method)
ProxyBasicAuthHandler (class in urllib.request)
ProxyDigestAuthHandler (class in urllib.request)
ProxyHandler (class in urllib.request)
ProxyType (in module weakref)
ProxyTypes (in module weakref)
pryear() (calendar.TextCalendar method)
ps1 (in module sys)
ps2 (in module sys)
pstats (module)
pstdev() (in module statistics)
pthread_getcpuclockid() (in module time)
pthread_kill() (in module signal)  PySet_
pthread_sigmask() (in module signal)  PySet_
threads  PySet_
pty  PySet_
    module  PySet_
pty (module)  PySet_
pull() (in module turtle)  PySet_
publicId (xml.dom.DocumentType attribute)  PySet_
PullDom (class in xml.dom.pulldom)  PySet_
punctuation (in module string)  PySet_
punctuation_chars (shlex.shlex attribute)  PySetOl
PurePath (class in pathlib)  PySigna
PurePath.anchor (in module pathlib)  PySlice_
PurePath.drive (in module pathlib)  PySlice_
PurePath.name (in module pathlib)  PySlice_
PurePath.parent (in module pathlib)  PySlice_
PurePath.parents (in module pathlib)  PySlice_
PurePath.parts (in module pathlib)  PySlice_
PurePath.root (in module pathlib)  PySlice_
PurePath.stem (in module pathlib)  PyState.
PurePath.suffix (in module pathlib)  PyState.
PurePath.suffixes (in module pathlib)  PyState.
PurePosixPath (class in pathlib)  PyStruc
PureProxy (class in smtpd)  PyStruc
PureWindowsPath (class in pathlib)  PyStruc
purge() (in module re)  PyStruc
Purpose.CLIENT_AUTH (in module ssl)  PyStruc
Purpose.SERVER_AUTH (in module ssl)  PyStruc
push() (asyncio.async_chat method)  PyStruc
    (code.InteractiveConsole method)  PyStruc
    (contextlib.ExitStack method)  PyStruc
push_alignment() (formatter.formatter method)  PyStruc
push_async_callback() (contextlib.AsyncExitStack method)  variable
push_async_exit() (contextlib.AsyncExitStack method) PySys_
push_font() (formatter.formatter method) PySys_
push_margin() (formatter.formatter method) PySys_
push_source() (shlex.shlex method) PySys_
push_style() (formatter.formatter method) PySys_
push_token() (shlex.shlex method) PySys_
push_with_producer() (asyncio.async_chat method) PySys_
pushbutton() (msilib.Dialog method) PySys_
put() (asyncio.Queue method) PySys_
     (multiprocessing.Queue method) PySys_
     (multiprocessing.SimpleQueue method) PySys_
     (queue.Queue method) PySys_
     (queue.SimpleQueue method) PySys_
put_nowait() (asyncio.Queue method) PySys_
     (multiprocessing.Queue method) PySys_
     (queue.Queue method) PySys_
     (queue.SimpleQueue method) PySys_
putch() (in module msvcrt) PEI
putenv() (in module os) PEI
putheader() (http.client.HTTPConnection method) PEI
putp() (in module curses) PEI
putrequest() (http.client.HTTPConnection method) PEI
putwch() (in module msvcrt) PEI
putwin() (curses.window method) PEI
pvariance() (in module statistics) PEI
pwd
     module PEI
pwd (module) PEI
pwd() (ftplib.FTP method) PEI
pwrite() (in module os) PEI
pwritev() (in module os) PEI
Py_ABS (C macro) PEI
Py_AddPendingCall (C function)

Py_AddPendingCall()

Py_AtExit (C function)

Py_BEGIN_ALLOW_THREADS (C macro)

Py_BLOCK_THREADS (C macro)

Py_buffer (C type)

Py_buffer.buf (C member)

Py_buffer.format (C member)

Py_buffer.internal (C member)

Py_buffer.itemsize (C member)

Py_buffer.len (C member)

Py_buffer.ndim (C member)

Py_buffer.obj (C member)

Py_buffer.readonly (C member)

Py_buffer.shape (C member)

Py_buffer.strides (C member)

Py_buffer.suboffsets (C member)

Py_BuildValue (C function)

Py_BufferWarningFlag (C variable)

Py_CHARMASK (C macro)

Py_CLEAR (C function)

py_compile (module)

PY_COMPILED (in module imp)

Py_CompileString (C function)

Py_CompileString(), [1], [2]

Py_CompileStringExFlags (C function)

Py_CompileStringFlags (C function)

[9],

Py_CompileStringObject (C function)

[17]

Py_complex (C type)

Py_DebugFlag (C variable)

Py_DecodeLocale (C function)

Py_DECREF (C function)

Py_DECREF()
Py_DontWriteBytecodeFlag (C variable)
Py_Ellipsis (C variable)
Py_EncodeLocale (C function)
Py_END_ALLOW_THREADS (C macro)
Py_EndInterpreter (C function)
Py_EnterRecursiveCall (C function)
Py_eval_input (C variable)
Py_Exit (C function)
Py_False (C variable)
Py_FatalError (C function)
Py_FatalError()
Py_FdIsInteractive (C function)
Py_file_input (C variable)
Py_Finalize (C function)
Py_FinalizeEx (C function)
Py_FinalizeEx(), [1], [2], [3], [4]
PY_FROZEN (in module imp)
Py_FrozenFlag (C variable)
Py_GetBuildInfo (C function)
Py_GetCompiler (C function)
Py_GetCopyright (C function)
Py_GetPath (C function)
Py_GetPrefix (C function)
Py_GetPath(), [1], [2]
Py_GetPlatform (C function)
Py_GetPrefix (C function)
Py_GetPrefix()
Py_GetProgramFullPath (C function)
Py_GetProgramFullPath()
Py_GetProgramName (C function)
Py_GetPythonHome (C function)[9],
Py_GetVersion (C function)  [17]
Py_HashRandomizationFlag (C variable)  PEI
Py_IgnoreEnvironmentFlag (C variable)  PEI
Py_INCREF (C function)  PEI
Py_INCREF()  PEI
Py_Initialize (C function)  PEI
Py_Initialize(), [1], [2], [3]  PEI
Py_InitializeEx (C function)  PEI
Py_InspectFlag (C variable)  PEI
Py_InteractiveFlag (C variable)  PEI
Py_IsInitialized (C function)  PEI
Py_IsInitialized()  PEI
Py_IsolatedFlag (C variable)  PEI
Py_LeaveRecursiveCall (C function)  [9]
Py_LegacyWindowsFSEncodingFlag (C variable)  PEI
Py_LegacyWindowsStdioFlag (C variable)  PEI
Py_Main (C function)  PEI
Py_MAX (C macro)  [9],
Py_MEMBER_SIZE (C macro)  PEI
Py_MIN (C macro)  PEI
Py_mod_create (C variable)  PEI
Py_mod_exec (C variable)  PEI
Py_NewInterpreter (C function)  PEI
Py_None (C variable)  PEI
Py_NoSiteFlag (C variable)  PEI
Py_NotImplemented (C variable)  PEI
Py_NoUserSiteDirectory (C variable)  PEI
py_object (class in ctypes)  PEI
Py_OptimizeFlag (C variable)  PEI
Py_PRINT_RAW  PEI
Py_QuietFlag (C variable)  PEI
Py_REFCNT (C macro)  PEI
Py_ReprEnter (C function)  PEI
Py_ReprLeave (C function)  PEI
Py_RETURN_FALSE (C macro)  
Py_RETURN_NONE (C macro)  
Py_RETURN_NOTIMPLEMENTED (C macro)  
Py_RETURN_RICHCOMPARE (C function)  
Py_RETURN_TRUE (C macro)  
Py_SetPath (C function)  
Py_SetPath()  
Py_SetProgramName (C function)  
Py_SetProgramName(), [1], [2], [3]  
Py_SetPythonHome (C function)  
Py_SetStandardStreamEncoding (C function)  
Py_single_input (C variable)  
Py_SIZE (C macro)  
PY_SOURCE (in module imp)  
PY_SSIZE_T_MAX  
Py_STRINGIFY (C macro)  
Py_TPFLAGS_BASE_EXC_SUBCLASS (built-in variable)  
Py_TPFLAGS_BASETYPE (built-in variable)  
Py_TPFLAGS_BYTES_SUBCLASS (built-in variable)  
Py_TPFLAGS_DEFAULT (built-in variable)  
Py_TPFLAGS_DICT_SUBCLASS (built-in variable)  
Py_TPFLAGS_HAVE_FINALIZE (built-in variable)  
Py_TPFLAGS_HAVE_GC (built-in variable)  
Py_TPFLAGS_HEAPTYPE (built-in variable)  
Py_TPFLAGS_LIST_SUBCLASS (built-in variable)  
Py_TPFLAGS_LONG_SUBCLASS (built-in variable)  
Py_TPFLAGS_READY (built-in variable)  
Py_TPFLAGS_READYING (built-in variable)  
Py_TPFLAGS_TUPLE_SUBCLASS (built-in variable)  
Py_TPFLAGS_TYPE_SUBCLASS (built-in variable)  
Py_TPFLAGS_UNICODE_SUBCLASS (built-in variable)  
PEP 373  
PEP 378  
PEP 380  
PEP 383  
PEP 384  
PEP 385  
PEP 389  
PEP 391  
PEP 392  
PEP 393  
PEP 395  
PEP 397  
PEP 398  
PEP 4  
PEP 405  
PEP 409  
PEP 411  
PEP 412  
PEP 414  
PEP 418  
PEP 420  
PEP 421  
PEP 424  
PEP 427  
PEP 428  
PEP 429  
PEP 432  
PEP 434  
PEP 435  
PEP 436  
PEP 441  
PEP 442
variable) PEI
Py_tracefunc (C type) PEI
Py_True (C variable) PEI
Py_tss_NEEDS_INIT (C macro) PEI
Py_tss_t (C type) PEI
Py_TYPE (C macro) PEI
Py_UCS1 (C type) [9],
Py_UCS2 (C type) PEI
Py_UCS4 (C type) [9],
Py_UNBLOCK_THREADS (C macro) PEI
Py_UnbufferedStdioFlag (C variable) PEI
Py_UNICODE (C type) PEI
Py_UNICODE_IS_HIGH_SURROGATE (C macro) PEI
Py_UNICODE_IS_LOW_SURROGATE (C macro) PEI
Py_UNICODE_IS_SURROGATE (C macro) PEI
Py_UNICODE_ISALNUM (C function) PEI
Py_UNICODE_ISALPHA (C function) PEI
Py_UNICODE_ISDECIMAL (C function) [9],
Py_UNICODE_ISDIGIT (C function) [17]
Py_UNICODE_ISLINEBREAK (C function) [25]
Py_UNICODE_ISRANGE (C function) PEI
Py_UNICODE_ISLOWER (C function) PEI
Py_UNICODE_ISNUMERIC (C function) PEI
Py_UNICODE_ISPRINTABLE (C function) PEI
Py_UNICODE_ISSPACE (C function) PEI
Py_UNICODE_JoinSurrogates (C function) PEI
Py_UNICODE_JoinSurrogates (C macro) [9],
Py_UNICODE_JoinSurrogates (C function) [17]
Py_UNICODE_TODECIMAL (C function) PEI
Py_UNICODE_TODIGIT (C function) PEI
Py_UNICODE_TOLOWER (C function) PEI
Py_UNICODE_TONUMERIC (C function) PEI
Py_UNICODE_TOTITLE (C function) PEI
Py_UNICODE_TOUPPER (C function) [9],
Py_UNICODE_TODIGIT (C function) PEI
Py_UNICODE_TONUMERIC (C function) PEI
Py_UNICODE_TOTITLE (C function) PEI
Py_UNICODE_TOUPPER (C function) PEI
Py_UNICODE_TODIGIT (C function) PEI
Py_UNICODE_TONUMERIC (C function) PEI
Py_UNICODE_TOTITLE (C function) PEI
Py_UNICODE_TOUPPER (C function) PEI
Py_UNICODE_TODIGIT (C function) PEI
Py_UNICODE_TONUMERIC (C function) PEI
Py_UNICODE_TOTITLE (C function) PEI
Py_UNICODE_TOUPPER (C function) PEI
Py_UNUSED (C macro)
Py_VaBuildValue (C function)
Py_VerboseFlag (C variable)
Py_VISIT (C function)
Py_XDECREF (C function)
Py_XDECREF()
Py_XINCREF (C function)
PyAnySet_Check (C function)
PyAnySet_CheckExact (C function)
PyArg_Parse (C function)
PyArg_ParseTuple (C function)
PyArg_ParseTuple()
PyArg_ParseTupleAndKeywords (C function)
PyArg_ParseTupleAndKeywords()
PyArg_UnpackTuple (C function)
PyArg_ValidateKeywordArguments (C function)
PyArg_VaParse (C function)
PyArg_VaParseTupleAndKeywords (C function)
PyArg_VaParseTupleAndKeywords()
PyASCIIObject (C type)
PyAsyncMethods (C type)
PyAsyncMethods.am_aiter (C member)
PyAsyncMethods.am_anext (C member)
PyAsyncMethods.am_await (C member)
PyBool_Check (C function)
PyBool_FromLong (C function)
PyBUF_ANY_CONTIGUOUS (C macro)
PyBUF_C_CONTIGUOUS (C macro)
PyBUF_CONTIG (C macro)
PyBUF_CONTIG_RO (C macro)
PyBUF_F_CONTIGUOUS (C macro)
PyBUF_FORMAT (C macro)
PyBUF_FULL (C macro)
PyBUF_FULL_RO (C macro)
PyBUF_INDIRECT (C macro)
PyBUF_ND (C macro), [1]  PEI
PyBUF_RECORDS (C macro)  PEI
PyBUF_RECORDS_RO (C macro)  PEI
PyBUF_SIMPLE (C macro)  PEI
PyBUF_STRIDED (C macro)  PEI
PyBUF_STRIDED_RO (C macro)  PEI
PyBUF_STRIDES (C macro)  Python
PyBUF_WRITABLE (C macro)  .py
PyBuffer_FillContiguousStrides (C function)  Python
PyBuffer_FillInfo (C function)  Python
PyBuffer_IsContiguous (C function)  Python
PyBuffer_Release (C function)  Python
PyBuffer_SizeFromFormat (C function)  Python
PyBuffer_ToContiguous (C function)  Python
PyBufferProcs  Python
    (C type)  Python
    PyBufferProcs.bf_getbuffer (C member)  python
    PyBufferProcs.bf_releasebuffer (C member)  test
PyByteArray_AS_STRING (C function)  Python
PyByteArray_AsString (C function)  Python
PyByteArray_Check (C function)  PEI
PyByteArray_CheckExact (C function)  PEI
PyByteArray_Concat (C function)  PEI
PyByteArray_FromObject (C function)  PEI
PyByteArray_GET_SIZE (C function)  PEI
PyByteArray_Resize (C function)  PEI
PyByteArray_Size (C function)  PEI
PyByteArray_Type (C variable)  PEI
PyByteArrayObject (C type)  [3], [4], [5]
PyBytes_AS_STRING (C function)  Python
PyBytes_AsString (C function)  Python
PyBytes_AsStringAndSize (C function)  Python
PyBytes_Check (C function)  [7]
PyBytes_CheckExact (C function)  PYTHON
PyBytes_Concat (C function)     PYTHON
PyBytes_ConcatAndDel (C function) PYTHON
PyBytes_FromFormat (C function) PYTHON
PyBytes_FromFormatV (C function) PYTHON
PyBytes_FromObject (C function)  PYTHON
PyBytes_FromString (C function)  PYTHON
PyBytes_FromStringAndSize (C function) [1], [2] PYTHON
PyBytes_GET_SIZE (C function)     PYTHON
PyBytes_Size (C function)        PYTHON
PyBytes_Type (C variable)        PYTHON
PyBytesObject (C type)           PYTHON
PyCallable_Check (C function)     PYTHON
PyCallIter_Check (C function)     PYTHON
PyCallIter_New (C function)       PYTHON
PyCallIter_Type (C variable)      PYTHON
PyCapsule (C type)                PYTHON
PyCapsule_CheckExact (C function) PYTHON
PyCapsule_Destructor (C type)      PYTHON
PyCapsule_GetContext (C function) PYTHON
PyCapsule_GetDestructor (C function) PYTHON
PyCapsule_GetName (C function)    PYTHON
PyCapsule_GetPointer (C function) PYTHON
PyCapsule_Import (C function)     PYTHON
PyCapsule_IsValid (C function)    PYTHON
PyCapsule_New (C function)        PYTHON
PyCapsule_SetContext (C function) PYTHON
PyCapsule_SetDestructor (C function) PYTHON
PyCapsule_SetName (C function)    PYTHON
PyCapsule_SetPointer (C function) PYTHON
PyCell_Check (C function)         Python
PyCell_GET (C function)           Python
PyCell_Get (C function)           Python
PyCell_New (C function)           Python
PyThread_create_key (C function)  Python
PyThread_delete_key (C function)  Python
PyThread_delete_key_value (C function) Python
PyThread_get_key_value (C function) Python
PyCell_SET (C function)  
PyCell_Set (C function)  
PyCell_Type (C variable)  
PyCellObject (C type)  
PyCFunction (C type)  
PyCFunctionWithKeywords (C type)  
PyClnValidationMode (class in py_compile)  
pyclbr (module)  
PyCode_Check (C function)  
PyCode_GetNumFree (C function)  
PyCode_New (C function)  
PyCode_NewEmpty (C function)  
PyCode_Type (C variable)  
PyCodec_BackslashReplaceErrors (C function)  
PyCodec_Decode (C function)  
PyCodec_Decoder (C function)  
PyCodec_Encode (C function)  
PyCodec_Encoder (C function)  
PyCodec_IgnoreErrors (C function)  
PyCodec_IncrementalDecoder (C function)  
PyCodec_IncrementalEncoder (C function)  
PyCodec_KnownEncoding (C function)  
PyCodec_LookupError (C function)  
PyCodec_NameReplaceErrors (C function)  
PyCodec_Register (C function)  
PyCodec_RegisterError (C function)  
PyCodec_ReplacementChars (C function)  
PyCodec_ReportEncodingError (C function)  
PyCodec_RegisterError (C function)  
PyCodec_SafeGetLastEncodingError (C function)  
PyCodec_SafeGetNextEncodingError (C function)  
PyCodec_StrictErrors (C function)  
PyCodec_XMLCharRefReplaceErrors (C function)  
PyCodeObject (C type)  
PyCompileError  
PyCompactUnicodeObject (C type)  
PyCompileError
PyCompilerFlags (C type)  PyTuple
PyComplex_AsCComplex (C function)  PyTuple
PyComplex_Check (C function)  PyTuple
PyComplex_CheckExact (C function)  PyTuple
PyComplex_FromCComplex (C function)  PyTuple
PyComplex_FromDoubles (C function)  PyTuple
PyComplex_ImagAsDouble (C function)  PyTuple
PyComplex_RealAsDouble (C function)  PyTuple
PyComplex_Type (C variable)  PyTuple
PyComplexObject (C type)  PyTuple
PyContext (C type)  PyTuple
PyContext_CheckExact (C function)  PyTuple
PyContext_ClearFreeList (C function)  PyTuple
PyContext_Copy (C function)  PyTuple
PyContext_CopyCurrent (C function)  PyType
PyContext_Enter (C function)  PyType
PyContext_Exit (C function)  PyType
PyContext_New (C function)  PyType
PyContext_Type (C variable)  PyType
PyContextToken (C type)  PyType
PyContextToken_CheckExact (C function)  PyType
PyContextToken_Type (C variable)  PyType
PyContextVar (C type)  PyType
PyContextVar_CheckExact (C function)  PyType
PyContextVar_Get (C function)  PyType
PyContextVar_Reset (C function)  PyType
PyContextVar_Set (C function)  PyType
PyContextVar_Type (C variable)  PyType
PyCoro_CheckExact (C function)  PyType
PyCoro_New (C function)  PyType
PyCoro_Type (C variable)  PyType
PyCoroObject (C type)  PyType
PyDate_Check (C function)  PyType
PyDate_CheckExact (C function)  PyTypeObject
PyDate_FromDate (C function)   PyTypeObject
PyDate_FromTimestamp (C function) PyTypeObject
PyDateTime_Check (C function)  PyTypeObject
PyDateTime_CheckExact (C function) PyTypeObject
PyDateTime_DATE_GET_HOUR (C function) PyTypeObject
PyDateTime_DATE_GET_MICROSECOND (C function) PyTypeObject
PyDateTime_DATE_GET_MINUTE (C function) PyTypeObject
PyDateTime_DATE_GET_SECOND (C function) PyTypeObject
PyDateTime_DELTA_GET_DAYS (C function) PyTypeObject
PyDateTime_DELTA_GET_MICROSECONDS (C function) PyTypeObject
PyDateTime_DELTA_GET_SECONDS (C function) PyTypeObject
PyDateTime_FromDateAndTime (C function) PyTypeObject
PyDateTime_FROM_TIMESTAMP (C function) PyTypeObject
PyDateTime_GET_DAY (C function) PyTypeObject
PyDateTime_TIME_GET_HOUR (C function) PyTypeObject
PyDateTime_TIME_GET_MICROSECOND (C function) PyTypeObject
PyDateTime_TIME_GET_MINUTE (C function) PyTypeObject
PyDateTime_TIME_GET_SECOND (C function) PyTypeObject
PyDateTime_TimeZone_UTC (C variable) PyTypeObject
PyDelta_Check (C function) PyTypeObject
PyDelta_CheckExact (C function) PyTypeObject
PyDelta_FromDSU (C function) PyTypeObject
PyDescr_IsData (C function) PyTypeObject
PyDescr_NewClassMethod (C function) PyTypeObject
PyDescr_NewGetSet (C function) PyTypeObject
PyDescr_NewMember (C function) PyTypeObject
PyDescr_NewMethod (C function) PyTypeObject
PyDescr_NewWrapper (C function) PyTypeObject
PyDict_Check (C function)  PyTypeObject
PyDict_CheckExact (C function)  PyTypeObject
PyDict_Clear (C function)  PyTypeObject
PyDict_ClearFreeList (C function)  PyTypeObject
PyDictContains (C function)  PyTypeObject
PyDict_Copy (C function)  PyTypeObject
PyDict_DelItem (C function)  PyTypeObject
PyDict_DelItemString (C function)  PyTZInfo
PyDict_GetItem (C function)  PyTZInfo
PyDict_GetItemString (C function)  PyTZInfo
PyDict_GetItemWithError (C function)  PyUnicc
PyDict_Items (C function)  PyUnicc
PyDict_Keys (C function)  PyUnicc
PyDict_Merge (C function)  PyUnicc
PyDict_MergeFromSeq2 (C function)  PyUnicc
PyDict_New (C function)  PyUnicc
PyDict_Next (C function)  PyUnicc
PyDict_SetDefault (C function)  PyUnicc
PyDict_SetItem (C function)  PyUnicc
PyDict_SetItemString (C function)  PyUnicc
PyDict_Size (C function)  PyUnicc
PyDict_Type (C variable)  PyUnicc
PyDict_Update (C function)  PyUnicc
PyDictValues (C function)  function
PyDictObject (C type)  PyUnicc
PyDictProxy_New (C function)  PyUnicc
PyDLL (class in ctypes)  PyUnicc
pydoc (module)  PyUnicc
PyErr_BadArgument (C function)  PyUnicc
PyErr_BadInternalCall (C function)  PyUnicc
PyErr_CheckSignals (C function)  function
PyErr_Clear (C function)  PyUnicc
PyErr_Clear(), [1]  PyUnicc
PyErr_ExceptionMatches (C function)  PyUnicc
PyUnicode_1BYTE_DATA (C function)
PyUnicode_1BYTE_KIND (C macro)
PyUnicode_2BYTE_DATA (C function)
PyUnicode_2BYTE_KIND (C macro)
PyUnicode_4BYTE_DATA (C function)
PyUnicode_4BYTE_KIND (C macro)
PyUnicode_AS_DATA (C function)
PyUnicode_AS_UNICODE (C function)
PyUnicode_AsASCIIString (C function)
PyUnicode_AsCharmapString (C function)
PyUnicode_AsEncodedString (C function)
PyUnicode_AsLatin1String (C function)
PyUnicode_AsMBCSString (C function)
PyUnicode_AsRawUnicodeEscapeString (C function)
PyUnicode_AsUCS4 (C function)
PyUnicode_AsUCS4Copy (C function)
PyUnicode_AsUnicode (C function)
PyUnicode_AsUnicodeAndSize (C function)
PyUnicode_AsUnicodeCopy (C function)
PyUnicode_AsUnicodeEscapeString (C function)
PyUnicode_AsUTF16String (C function)
PyUnicode_AsUTF32String (C function)
PyUnicode_AsUTF8 (C function)
function
PyErr_SetImportError (C function)
PyErr_SetImportErrorSubclass (C function)
PyErr_SetInterrupt (C function)
PyErr_SetNone (C function)
PyErr_SetObject (C function)
PyErr_SetString (C function)
PyErr_SyntaxLocation (C function)
PyErr_SyntaxLocationEx (C function)
PyErr_SyntaxLocationObject (C function)
PyErr_WarnEx (C function)
PyErr_WarnExplicit (C function)
PyErr_WarnExplicitObject (C function)
PyErr_WarnFormat (C function)
PyErr_WriteUnraisable (C function)
PyEval_AcquireLock (C function)
PyEval_AcquireThread (C function)
PyEval_AcquireThread()
PyEval_EvalCode (C function)
PyEval_EvalCodeEx (C function)
PyEval_EvalFrame (C function)
PyEval_EvalFrameEx (C function)
PyEval_GetBuiltins (C function)
Index – Q

qiflush() (in module curses)
QName (class in xml.etree.ElementTree)
qsize() (asyncio.Queue method)
    (multiprocessing.Queue method)
    (queue.Queue method)
    (queue.SimpleQueue method)
qualified name
quantize() (decimal.Context method)
    (decimal.Decimal method)
QueryInfoKey() (in module winreg)
QueryReflectionKey() (in module winreg)
QueryValue() (in module winreg)
QueryValueEx() (in module winreg)
Queue (class in asyncio)
    (class in multiprocessing)
    (class in queue)
queue (module)
    (sched.scheduler attribute)
Queue()
    (multiprocessing.managers.SyncManager method)
QueueEmpty
QueueFull
QueueHandler (class in logging.handlers)
QueueListener (class in logging.handlers)
quick_ratio()
    (difflib.SequenceMatcher method)
quit (built-in variable)
    (pdb command)
    (ftplib.FTP method)
    (nntplib.NNTP method)
    (poplib.POP3 method)
    (smtplib.SMTP method)
quopri (module)
quote() (in module email.utils)
    (in module shlex)
    (in module urllib.parse)
QUOTE_ALL (in module csv)
quote_from_bytes()
    (module urllib.parse)
QUOTE_MINIMAL (module csv)
QUOTE_NONE (module csv)
QUOTE_NONNUM (in module csv)
quote_plus() (in module urllib.parse)
quoteattr() (in module xml.sax.saxutils)
quotechar (csv.Dialect attribute)
quoting (csv.Dialect attribute)
quotes (shlex.attribute)
quoted-printable encoding
Index – R

r" raw string literal
r' raw string literal
R_OK (in module os)
radians() (in module math)
    (in module turtle)
RadioButtonGroup (class in msilib)
radiogroup() (msilib.Dialog method)
radx() (decimal.Context method)
    (decimal.Decimal method)
RADIXCHAR (in module locale)
raise statement, [1]
raise (2to3 fixer)
raise an exception
raise_on_defect (email.policy.Policy attribute)
RAISE_VARARGS (opcode)
raising exception
RAND_add() (in module ssl)
RAND_bytes() (in module ssl)
RAND_egd() (in module ssl)
RAND_pseudo_bytes() (in module ssl)
RAND_status() (in module ssl)
randbelow() (in module secrets)
randbits() (in module secrets)
randint() (in module random)
random (module)
random() (in module random)
randrange() (in module random)
range
    built-in function
    object
range (built-in class)
RARROW (in module token)
ratecv() (in module audioop)
ratio() (difflib.SequenceMatcher method)
Rational (class in numbers)
raw (io.BufferedIOBase attribute)
raw string
raw() (in module curses)
raw_data_manager (in module email.contentmanager)
raw_decode() (json.JSONDecoder method)
raw_input (2to3 fixer)
raw_input() (code.InteractiveConsole method)
RawArray() (in module multiprocessing.sharedctypes)
RawConfigParser (class in configparser)
RawDescriptionHelpFormatter (class in argparse)
RawIOBase (class in io)
RawPen (class in turtle)
RawTextHelpFormatter (class in argparse)
RawTurtle (class in turtle)
RawValue() (in module multiprocessing.sharedctypes)
RBRACE (in module token)
rcttos (smtpd.SMTPChannel attribute)
re
    module, [1]
re (module)
    (re.Match attribute)
read() (asyncio.StreamReader method)
read_history_file() (in module readline)
read_init_file() (in module readline)
read_lazy() (telnetlib.Telnet method)
read_mime_types() (in module mimetypes)
READ_RESTRICTED
read_sb_data() (telnetlib.Telnet method)
read_some() (telnetlib.Telnet method)
read_string() (configparser.ConfigParser method)
read_text() (in module importlib.resources)
    (pathlib.Path method)
read_token() (shlex.shlex method)
read_until() (telnetlib.Telnet method)
read_very_eager() (telnetlib.Telnet method)
read_very_lazy() (telnetlib.Telnet method)
read_windows_registry() (mimetypes.MimeTypes method)
READABLE (in module tkinter)
readable() (asyncore.dispatcher method)
    (io.IOBase method)
readall() (io.RawIOBase method)
reader() (in module csv)
ReadError
readexactly() (asyncio.StreamReader method)
readfp() (configparser.ConfigParser method)
    (mimetypes.MimeTypes method)
readframes() (aifc.aifc method)
    (sunau.AU_read method)
    (wave.Wave_read method)
readinto() (http.client.HTTPResponse method)
    (io.BufferedIOBase method)
    (io.RawIOBase method)
readinto1() (io.BufferedIOBase method)
    (io.BytesIO method)
readline (module)
readline() (asyncio.StreamReader method)
    (codecs.StreamReader method)
    (distutils.text_file.TextFile method)
    (imaplib.IMAP4 method)
    (io.IOBase method)
    (io.TextIOBase method)
    (mmap.mmap method)
readlines() (codecs.StreamReader method)
    (distutils.text_file.TextFile method)
    (io.IOBase method)
readlink() (in module os)
readmodule() (in module pyclbr)
readmodule_ex() (in module pyclbr)
READONLY
readonly (memoryview attribute)
readPlist() (in module plistlib)
readPlistFromBytes() (in module plistlib)
ReadTransport (class in asyncio)
readuntil() (asyncio.StreamReader method)
readv() (in module os)
ready() (multiprocessing.poolAsyncResult method)
Real (class in numbers)
real (numbers.Complex attribute)
Real Media File Format
real_max_memuse (in module test.support)
real_quick_ratio() (difflib.SequenceMatcher method)
realloc()
realpath() (in module os.path)
REALTIME_PRIORITY_CLASS (in module subprocess)
reap_children() (in module test.support)
reap_threads() (in module test.support) RFC 1014
reason (http.client.HTTPResponse attribute) RFC 1123
   (UnicodeError attribute) RFC 1321
   (ssl.SSL_ERROR attribute) RFC 1422
   (urllib.error.HTTPError attribute) RFC 1521
   (urllib.error.URLError attribute) RFC 1522
reattach() (tkinter.ttk.Treeview method) RFC 1524
rebinding RFC 1730
   name RFC 1738
reccontrols() (ossaudiodev.oss_mixer_device method) RFC 1750
received_data (smtplib.SMTPChannel attribute) RFC 1808
received_lines (smtplib.SMTPChannel attribute) RFC 1832
recent() (imaplib.IMAP4 method) RFC 1869
reconfigure() (io.TextIOWrapper method) RFC 1870
record_original_stdout() (in module test.support) RFC 1939
records (unittest.TestCase attribute) RFC 2033
rect() (in module cmath) RFC 2045
rectangle() (in module curses.textpad) RFC 2046
RecursionError [9], [10], [1]
recursive_repr() (in module reprlib) RFC 2047
recv() (asyncore.dispatcher method) RFC 2060
   (multiprocessing.connection.Connection method) RFC 2104
   (socket.socket method) RFC 2109
recv_bytes() [9], [10], [1]
   (multiprocessing.connection.Connection method) RFC 2183
recv_bytes_into() [9], [10], [1]
   (multiprocessing.connection.Connection method) RFC 2231
RFC 2295
RFC 2342
RFC 2368
recv_into() (socket.socket method) RFC 2373
recvfrom() (socket.socket method) RFC 2396
recvfrom_into() (socket.socket method) RFC 2397
recvmsg() (socket.socket method) RFC 2449
recvmsg_into() (socket.socket method) RFC 2487
redirect_request() RFC 2518
(urslib.request.HTTPRedirectHandler method) RFC 2595
redirect_stderr() (in module contextlib) RFC 2616
redirect_stdout() (in module contextlib) [9]
redisplay() (in module readline) RFC 2732
redrawln() (curses.window method) RFC 2774
redrawwin() (curses.window method) RFC 2818
reduce (2to3 fixer) RFC 2821
reduce() (in module functools) RFC 2822
ref (class in weakref) [9], [10], [1]
refcount_test() (in module test.support) RFC 2964
reference RFC 2965
          attribute [9], [10], [17], [18],
reference count
reference counting RFC 2980
ReferenceError, [1] RFC 3056
ReferenceType (in module weakref) RFC 3171
refold_source (email.policy.EmailPolicy attribute) RFC 3207
refresh() (curses.window method) RFC 3229
REG_BINARY (in module winreg) RFC 3280
REG_DWORD (in module winreg) RFC 3330
REG_DWORD_BIG_ENDIAN (in module winreg) RFC 3339
REG_DWORD_LITTLE_ENDIAN (in module winreg) RFC 3454
REG_EXPAND_SZ (in module winreg) RFC 3490
REG_FULL_RESOURCE_DESCRIPTOR (in module winreg) RFC 3490
RFC 3492
RFC 3493
RFC 3501
RFC 3542
REG_LINK (in module winreg) RFC 3548
REG_MULTI_SZ (in module winreg) RFC 3659
REG_NONE (in module winreg) RFC 3879
REG_QWORD (in module winreg) RFC 3927
REG_QWORD_LITTLE_ENDIAN (in module winreg) RFC 3977
REG_RESOURCE_LIST (in module winreg) RFC 4086
REG_RESOURCE_REQUIREMENTS_LIST (in module winreg) RFC 4122
REG_SZ (in module winreg) RFC 4193
register() (abc.ABCMeta method) RFC 4217
    (in module atexit) RFC 4291
    (in module codecs) RFC 4380
    (in module faulthandler) RFC 4627
    (in module webbrowser) RFC 4642
    (multiprocessing.managers.BaseManager method) RFC 4918
    (select.devpoll method) RFC 5161
    (select.epoll method) RFC 5233
    (select.poll method) RFC 5246
    (selectors.BaseSelector method) RFC 5280
register_adapter() (in module sqlite3) RFC 5321
register_archive_format() (in module shutil) RFC 5322
register_at_fork() (in module os) [9], [10],
register_converter() (in module sqlite3) [17], [18],
register_defect() (email.policy.Policy method) RFC 5424
register_dialect() (in module csv) RFC 5424
register_error() (in module codecs) RFC 5735
register_function()
    (xmlrpc.server.CGIXMLRPCRequestHandler method) RFC 5929
    (xmlrpc.server.SimpleXMLRPCServer method) RFC 6066
register_instance() RFC 6531
(xmlrpc.server.CGIXMLRPCRequestHandler method) RFC 6532
(xmlrpc.server.SimpleXMLRPCServer method) RFC 6585
register_introspection_functions() RFC 6855
(xmlrpc.server.CGIXMLRPCRequestHandler method) RFC 6856
(register_multicall_functions() RFC 7159
(xmlrpc.server.CGIXMLRPCRequestHandler method) RFC 7230
(register_namespace() (in module xml.etree.ElementTree) RFC 7231
register_optionflag() (in module doctest) RFC 7232
register_shape() (in module turtle) RFC 7233
register_unpack_format() (in module shutil) RFC 7235
registerDOMImplementation() (in module xml.dom) RFC 7238
registerResult() (in module unittest) RFC 7239
regular package RFC 7301
regular package RFC 7305
relative RFC 7405
relative
URL RFC 7540
import RFC 7693
relative_to() (pathlib.PurePath method) RFC 7914
release() (_thread.lock method) RFC 821,
(asyncio.Condition method) RFC 822,
(asyncio.Lock method) RFC 854,
(asyncio.Semaphore method) RFC 959,
(in module platform) RFC 977
/rfc2109 (http.cookiejar.Cookie attribute)
/rfc2109_as_netscape (http.cookiejar.DefaultCookiePolicy attribute)
/rfc2965 (http.cookiejar.CookiePolicy attribute)
/rfc822_escape
/rfc822 (in module distutils.util)
/rfc4122 (in module uuid)
rfile (http.server.BaseHTTPRequestHandler attribute)
rename() (ftplib.FTP method)  rotation_filename
       (imaplib.IMAP4 method) (logging.handlers.RotatingFileHandler method)
       (in module os)  rotator (logging.handlers.RotatingFileHandler attribute)
       (pathlib.Path method)  round
renames (2to3 fixer)  round() (built-in function)
renames() (in module os)
reopenIfNecessary()  ROUND_05UP
       (logging.handlers.WatchedFileHandler method)
reorganize() (dbm.gnu.gdbm method)  ROUND_CEILING
repeat() (in module itertools)  ROUND_DOWN
       (in module timeit)  ROUND_FLOOR
       (timeout.Timer method)  ROUND_HALF_UP
repetition  ROUND_HALF_UP
     operation  ROUND_HALF_DOWN
replace() (bytearray method)  ROUND_UP (in module decimal)
     (bytes method)  Rounded (class in decimal)
     (curses.panel.Panel method)  Row (class in sqlite3)
     (datetime.date method)  row_factory (sqlite3.Connection attribute)
     (datetime.datetime method)  rowcount (sqlite3.Cursor attribute)
     (datetime.time method)  RPAR (in module token)
     (in module dataclasses)  rpartition() (bytearray method)
     (in module os)  (bytes method)
     (inspect.Parameter method)  (str method)
     (inspect.Signature method)  rpc_paths
     (pathlib.Path method)  (xmlrpc.server.SimpleXMLRPCServer attribute)
     (str method)  rpop() (poplib.POP3 method)
replace_errors() (in module codecs) rset() (poplib.POP3 method)
replace_header()  rshift() (in module operator)
       (email.message.EmailMessage method)  rsplit() (bytearray method)
       (email.message.Message method)  (bytes method)
replace_history_item() (in module readline)  (str method)
replace_whitespace  RSQB (in module textwrap.TextWrapper attribute)
replaceChild() (xml.dom.Node method)

rstrip() (bytearray method)
(bytes method)
(str method)

rt() (in module turtle)
RTLD_DEEPBIND (in module os)
RTLD_GLOBAL (in module os)
RTLD_LAZY (in module os)
RTLD_LOCAL (in module os)
RTLD_NODELETE (in module os)
RTLD_NOLOAD (in module os)
RTLD_NOW (in module os)
ruler (cmd.Cmd attribute)
run (pdb command)
Run script
run() (bdb.Bdb method)
(contextvars.Context method)
(distutils.command method)
doctest.DoctestRunner (in module asyncio)
doctest.DoctestRunner (in module pdb)
doctest.DoctestRunner (in module profile)
doctest.DoctestRunner (in module subprocess)
doctest.DoctestRunner (in module multiprocessing)
doctest.DoctestRunner (in module pdb)
doctest.DoctestRunner (in module profile)
doctest.DoctestRunner (in module sched)
doctest.DoctestRunner (in module test.support)
doctest.DoctestRunner (in module threading)
doctest.DoctestRunner (in module trace)
doctest.DoctestRunner (in module unittest)
doctest.DoctestRunner (in module unittest)
doctest.DoctestRunner (in module unittest)
doctest.DoctestRunner (in module wsgiref)
run_coroutine_
run_docstring_examples()
run_doctest()
run_forever()
run_in_executor()
run_in_subinterp()
run_module()
run_path()
run_python_unittest()
run_script()
run_setup()
run_unittest()
run_until_complete()
run_with_locale()
run_with_tz()
runcall()
runcode()
runcx()
runcall()
runfunc()
running()
runtime_library_dir_option()
Index – S

S (in module re)
S_ENFMT (in module stat)
S_IEXEC (in module stat)
S_IFBLK (in module stat)
S_IFCHR (in module stat)
S_IFDIR (in module stat)
S_IFDOOR (in module stat)
S_IFIFO (in module stat)
S_IFLINK (in module stat)
S_IFMT() (in module stat)
S_IFPORT (in module stat)
S_IFREG (in module stat)
S_IFSOCK (in module stat)
S_IFWHT (in module stat)
S_IMODE() (in module stat)
S_IREAD (in module stat)
S_IRGRP (in module stat)
S_IROTH (in module stat)
S_IRUSR (in module stat)
S_IRUSR (in module stat)
S_IRWXR (in module stat)
S_IRWXR (in module stat)
S_IRWXU (in module stat)
S_IRWXU (in module stat)
S_ISBLK() (in module stat)
S_ISCHR() (in module stat)
S_ISDIR() (in module stat)
S_ISDOOR() (in module stat)
S_ISFIFO() (in module stat)
S_ISGID (in module stat)
S_ISLNK() (in module stat)
S_ISPORT() (in module stat)
S_ISREG() (in module stat)

SimpleHTTP (class in http.server)
SimpleNamespace (class in types)
SimpleQueue (class in multiprocessing)
SimpleXMLRPCRequestHandler (class in xmlrpc.server)
SimpleXMLRPCServer (class in xmlrpc.server)
sin() (in module math)
single dispatch
single disse
singleton
tuple
sinh() (in module math)
SIO_KEEPA
SIO_LOOPE
socket
SIO_RCVAL
site (module)
site command
--user-b
--user-s
site-package
directory
sitecustomize
module
S_ISSOCK() (in module stat)
S_ISUID (in module stat)
S_ISVTX (in module stat)
S_ISWHT() (in module stat)
S_IWGRP (in module stat)
S_IWOTH (in module stat)
S_IWRITE (in module stat)
S_IWUSR (in module stat)
S_IXGRP (in module stat)
S_IXOTH (in module stat)
S_IXUSR (in module stat)
safe (uuid.SafeUUID attribute)
safe_substitute() (string.Template method)
SafeChildWatcher (class in asyncio)
saferepr() (in module pprint)
SafeUUID (class in uuid)
same_files (filecmp.dircmp attribute)
same_quantum() (decimal.Context method)
samefile() (in module os.path)
SafeFileError
sameopenfile() (in module os.path)
samestat() (in module os.path)
sample() (in module random)
save() (http.cookiejar.FileCookieJar method)
SAVEDCWD (in module test.support)
SaveKey() (in module winreg)
SaveSignals (class in test.support)
savetty() (in module curses)
SAX2DOM (class in xml.dom.pulldom)
SAXException
SAXNotRecognizedException
SAXNotSupportedException
sixtofour (ipaddress.IPv6Address attribute)
size (struct.Struct attribute)
size (tarfile.TarInfo attribute)
size_diff (tracemalloc.StatisticDiff attribute)
SIZE_MAX
Sized (class in collections.abc)
sized (class in typing)
sizof() (in module ctypes)
skip() (chunk.Chunk method)
skipUnless() (in module unittest)
SLASH (in module token)
SLASHEQUAL (in module token)
slave() (nntplib.NNTP method)
sleep() (in module asyncio)
sleep() (in module time)
SkipTest
skiplf() (in module http.client)
skipinitialspace (csv.Dialect attribute)
skipped (unittest.TestResult attribute)
skippedEntity() (xml.sax.handler.ContentHandler method)
SlapChain
SLEHEQU
SAXParseException
scaleb() (decimal.Context method)
   (decimal.Decimal method)
scandir() (in module os)
scanf()
sched (module)
SCHED_BATCH (in module os)
SCHED_FIFO (in module os)
sched_get_priority_max() (in module os)
sched_get_priority_min() (in module os)
sched_getaffinity() (in module os)
sched_getparam() (in module os)
sched_getscheduler() (in module os)
SCHED_IDLE (in module os)
SCHED_OTHER (in module os)
sched_param (class in os)
sched_priority (os.sched_param attribute)
SCHED_RESET_ON_FORK (in module os)
SCHED_RR (in module os)
sched_rr_get_interval() (in module os)
sched_setaffinity() (in module os)
sched_setparam() (in module os)
sched_setscheduler() (in module os)
SCHED_SPORADIC (in module os)
sched_yield() (in module os)
scheduler (class in sched)
schema (in module msilib)
scope, [1]
Screen (class in turtle)
screensize() (in module turtle)
script_from_examples() (in module doctest)
scroll() (curses.window method)
ScrolledCanvas (class in turtle)
scrollok() (curses.window method)
scrypt() (in module hashlib)

sdterr

stdin stdout

seal() (in module unittest.mock)

search

path, module, [1], [2], [3], [4], [5], [6], [7]

search() (imaplib.IMAP4 method)

(re.Pattern method)

second (datetime.datetime attribute)

(datetime.time attribute)

seconds since the epoch

secrets (module)

SECTCRE (configparser.ConfigParser attribute)

sections() (configparser.ConfigParser method)

secure (http.cookiejar.Cookie attribute)

secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512

Secure Sockets Layer

security

CGI

see() (tkinter.ttk.Treeview method)

seed() (in module random)

seek() (chunk.Chunk method)

(io.IOBase method)

(io.TextIOBase method)

(mmap.mmap method)

SEEK_CUR (in module os)

SEEK_END (in module os)

SEEK_SET (in module os)

seekable() (io.IOBase method)

seen_greeting (smtpd.SMTPChannel attribute)

Select (class in tkinter.tix)

select (module)
select() (imaplib.IMAP4 method)
  (in module select)
  (selectors.BaseSelector method)
  (tkinter.ttk.Notebook method)
selected_alpn_protocol()  (ssl.SSLSocket method)
selected_npn_protocol()  (ssl.SSLSocket method)
selection() (tkinter.ttk.Treeview method)
selection_add() (tkinter.ttk.Treeview method)
selection_remove() (tkinter.ttk.Treeview method)
selection_set() (tkinter.ttk.Treeview method)
selection_toggle() (tkinter.ttk.Treeview method)
selector (urllib.request.Request.Request attribute)
SelectorEventLoop (class in asyncio)
SelectorKey (class in selectors)
selectors (module)
SelectSelector (class in selectors)
Semaphore (class in asyncio)
  (class in multiprocessing)
  (class in threading)
Semaphore()
  (multiprocessing.managers.SyncManager method)
semaphores, binary
SEMI (in module token)
send() (asyncore.dispatcher method)
  (coroutine method)
  (generator method)
  (http.client.HTTPConnection method)
  (imaplib.IMAP4 method)
  (logging.handlers.DatagramHandler method)
  (logging.handlers.SocketHandler method)
(multiprocessing.connection.Connection method)
(socket.socket method)
send_bytes()
(multiprocessing.connection.Connection method)
send_error()
(http.server.BaseHTTPRequestHandler method)
send_flowing_data() (formatter.writer method)
send_header()
(http.server.BaseHTTPRequestHandler method)
send_hor_rule() (formatter.writer method)
send_label_data() (formatter.writer method)
send_line_break() (formatter.writer method)
send_literal_data() (formatter.writer method)
send_message() (smtplib.SMTP method)
send_paragraph() (formatter.writer method)
send_response()
(http.server.BaseHTTPRequestHandler method)
send_response_only()
(http.server.BaseHTTPRequestHandler method)
send_signal()
(asyncio.asncio.subprocess.Process method)
    (asyncio.SubprocessTransport method)
    (subprocess.Popen method)
sendall() (socket.socket method)
sendcmd() (ftplib.FTP method)
sendfile() (asyncio.loop method)
    (in module os)
    (socket.socket method)
    (wsgiref.handlers.BaseHandler method)
SendfileNotAvailableError
sendmail() (smtplib.SMTP method)
sendmsg() (socket.socket method)
sendmsg_afalg() (socket.socket method)
sendto() (asyncio.DatagramTransport method)
    (socket.socket method)
sentinel (in module unittest.mock)
    (multiprocessing.Process attribute)
sep (in module os)
    (str method)

sequence
    item
    iteration
    object, [1], [2], [3], [4], [5], [6], [7], [8]
types, immutable
types, mutable
types, operations on, [1]
Sequence (class in collections.abc)
    (class in typing)
sequence (in module msilib)
sequence2st() (in module parser)
SequenceMatcher (class in difflib), [1]
serializing
    objects
serve_forever() (asyncio.Server method)
    (socketserver.BaseServer method)
server
    WWW, [1]
Server (class in asyncio)
server (http.server.BaseHTTPRequestHandler attribute)
server_activate() (socketserver.BaseServer method)
server_address (socketserver.BaseServer attribute)
server_bind() (socketserver.BaseServer method)
server_close() (socketserver.BaseServer method)

split_quoted()
    (in module distutils.util)
splitdrive() (in module os.path)
splitext() (in module os.path)
splitlines() (in module bytes)

sequence
    item
    iteration
    object, [1], [2], [3], [4], [5], [6], [7], [8]
types, immutable
types, mutable
types, operations on, [1]
Sequence (class in collections.abc)
    (class in typing)
sequence (in module msilib)
sequence2st() (in module parser)
SequenceMatcher (class in difflib), [1]
serializing
    objects
serve_forever() (asyncio.Server method)
    (socketserver.BaseServer method)
server
    WWW, [1]
Server (class in asyncio)
server (http.server.BaseHTTPRequestHandler attribute)
server_activate() (socketserver.BaseServer method)
server_address (socketserver.BaseServer attribute)
server_bind() (socketserver.BaseServer method)
server_close() (socketserver.BaseServer method)

SplitResult (class in urllib.parse)
SplitResultBytes (class in urllib.parse)
SpooledTemporaryFile () (in module tempfile)
sprintf-style formatting
spwd (module)
sqlite3 (module)
sqrt() (decimal.Decimal method)
    (in module cmath)
    (in module math)
SSL
ssl (module)
SSL_CERT
SSL_CERT
SSL_SERVER
ssl_version (ftplib.FTP_TLS attribute)
SSLCertVerificationError
SSLContext (class in ssl)
SSLEOFError
SSLError
SSLErrorNumber (class in ssl)
SSLObject (class in ssl)
sslsocket_class (ssl.SSLContext attribute)
method)
server_hostname (ssl.SSLSocket attribute) SSLSyscallError
server_side (ssl.SSLSocket attribute) SSLv3 (ssl.TLSVersion attribute)
server_software (wsgiref.handlers.BaseHandler attribute) SSLWantReadError
server_version SSLWantWriteError
(ssl.SSLSocket attribute)
ServerProxy (class in xmlrpc.client)
    service_actions() (socketserver.BaseServer method)
        (http.server.BaseHTTPRequestHandler attribute)
            (http.server.SimpleHTTPRequestHandler attribute)
            ServerProxy (class in xmlrpc.client)

set (built-in class)
Set (class in collections.abc)
    (class in typing)
    Set Breakpoint
    set type
        object
    set() (asyncio.Event method)
        (configparser.ConfigParser method)
        (configparser.RawConfigParser method)
        (contextvars.ContextVar method)
        (http.cookies.Morsel method)
        (osaudiodev.oss_mixer_device method)
        (test.support.EnvironmentVarGuard method)
        (threading.Event method)

session (ssl.SSLSocket attribute)
    session_reused (ssl.SSLSocket attribute)
        session_stats() (ssl.SSLContext method)

st() (in module turtle)
st2list() (in module parser)
st2tuple() (in module parser)
st_atime (os.stat_result attribute)
st_atime_ns (os.stat_result attribute)
st_birthtime (os.stat_result attribute)
st_blocks (os.stat_result attribute)
st_cftime (os.stat_result attribute)
st_ctime (os.stat_result attribute)
st_ctime_ns (os.stat_result attribute)
st_dev (os.stat_result attribute)
st_file_attributes (os.stat_result attribute)
st_flags (os.stat_result attribute)
st_fstype (os.stat_result attribute)
st_gen (os.stat_result attribute)
st_gid (os.stat_result attribute)
st_dev (os.stat_result attribute)
st_mode (os.stat_result attribute)
st_mtime (os.stat_result attribute)
st_mtime_ns (os.stat_result attribute)
st_nlink (os.stat_result attribute)
st_ino (os.stat_result attribute)
st_blocks (os.stat_result attribute)
st_dev (os.stat_result attribute)
st_dev (os.stat_result attribute)
st_mode (os.stat_result attribute)
st_mtime (os.stat_result attribute)
st_mtime_ns (os.stat_result attribute)
st_nlink (os.stat_result attribute)
st_ino (os.stat_result attribute)
st_dev (os.stat_result attribute)
ST_ATIME (in module time)
ST_CTIME (in module time)
ST_DEV (in module stat)
ST_GID (in module stat)
ST_INO (in module stat)
ST_MTIME (in module stat)
ST_NLINK (in module stat)
(tkinter.ttk.Combo method)
(tkinter.ttk.Spinbox method)
(tkinter.ttk.Treeview method)
(xml.etree.ElementTree.Element method)

SET_ADD (opcode)
set_all()
set_allowed_domains()
(http.cookiejar.DefaultCookiePolicy method)
set_alpn_protocols() (ssl.SSLContext method)
set_app() (wsgiref.simple_server.WSGIServer method)
set_asyncgen_hooks() (in module sys)
set_authorizer() (sqlite3.Connection method)
set_auto_history() (in module readline)
set_blocked_domains()
(http.cookiejar.DefaultCookiePolicy method)
set_blocking() (in module os)
set_boundary() (email.message.EmailMessage method)
(set.email.message.Message method)
set_break() (bdb.Bdb method)
set_charset() (email.message.Message method)
set_child_watcher()
(asyncio.AbstractEventLoopPolicy method)
(in module asyncio)
set_children() (tkinter.ttk.Treeview method)
set_ciphers() (ssl.SSLContext method)
set_completer() (in module readline)
set_completer_delims() (in module readline)
set_completion_display_matches_hook() (in module readline)
set_content()
(email.contentmanager.ContentManager method)

st_nlink (os.stat_result attribute)
st_rdev (os.stat_result attribute)
st_rsize (os.stat_result attribute)
ST_SIZE (in module stat)
st_size (os.stat_result attribute)
st_type (os.stat_result attribute)
stack
stack viewer
stack() (in module traceback)
stack_effect()
stack_size() (in module threading)
stackable
stacks
StackSummary (class in traceback)
stamp() (in module turtle)
standard
standard C
standard input
standard_b64decode() (in module base64)
standard_b64encode() (in module base64)
standarderror (2to3 fixer)
standend() (curses.window method)
standout() (curses.window method)
STAR (in module token)
STAREQUAL (in module token)
starmap() (in module itertools)
(asyncio.AbstractEventLoopPolicy method)
  (in module asyncio)
set_event_loop_policy() (in module asyncio)
set_exception() (asyncio.Future method)
  (concurrent.futures.Future method)
set_exception_handler() (asyncio.loop method)
set_executable() (in module multiprocessing)
set_executables() (distutils.ccompiler.CCompiler method)
set_flags() (mailbox.MaildirMessage method)
  (mailbox.MMDFMessage method)
  (mailbox.mboxMessage method)
set_from() (mailbox.mboxMessage method)
  (mailbox.MMDFMessage method)
set_handle_inheritable() (in module os)
set_history_length() (in module readline)
set_include_dirs() (distutils.ccompiler.CCompiler method)
set_info() (mailbox.MaildirMessage method)
set_inheritable() (in module os)
  (socket.socket method)
set_labels() (mailbox.BabylMessage method)
set_last_error() (in module ctypes)
set_libraries() (distutils.ccompiler.CCompiler method)
set_library_dirs() (distutils.ccompiler.CCompiler method)
set_link_objects() (distutils.ccompiler.CCompiler method)
set_literal (2to3 fixer)
set_loader() (in module importlib.util)
set_match_tests() (in module test.support)
set_memlimit() (in module test.support)
set_next() (bdb.Bdb method)

startElement()
  (xml.sax.handler.ContentHandler method)
STARTF_USESHOWWINDOW	 (in	 module 
subprocess)
STARTF_USESTDHANDLES	 (in	 module 
subprocess)
startfile() (in 
StartNamesp
startPrefixMi
startswith() (b
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
startfile()
StartNamespaceDeclHandler()
startPrefixM
startswith()
startTest() (u
startTestRun
starttls() (im
STARTUPIN
stat
module
stat (module
stat() (in mod
(nntplib.
set_nonstandard_attr()  (http.cookiejar.Cookie method)
set_npn_protocols()  (ssl.SSLContext method)
set_ok()  (http.cookiejar.CookiePolicy method)
set_option_negotiation_callback()  (telnetlib.Telnet method)
set_output_charset()  (gettext.NullTranslations method)
set_package()  (in module importlib.util)
set_param()  (email.message.EmailMessage method)
  (email.message.Message method)
set_pasv()  (ftplib.FTP method)
set_payload()  (email.message.Message method)
set_policy()  (http.cookiejar.CookieJar method)
set_position()  (xdrlib.Unpacker method)
set_pre_input_hook()  (in module readline)
set_progress_handler()  (sqlite3.Connection method)
set_protocol()  (asyncio.BaseTransport method)
set_proxy()  (urllib.request.Request method)
set_python_build()  (in module distutils.sysconfig)
set_quit()  (bdb.Bdb method)
set_recsrc()  (ossaudiodev.oss.mixer_device method)
set_result()  (asyncio.Future method)
  (concurrent.futures.Future method)
set_return()  (bdb.Bdb method)
set_running_or_notify_cancel()  (concurrent.futures.Future method)
set_runtime_library_dirs()  (distutils.ccompiler.CCompiler method)
set_seq1() (difflib.SequenceMatcher method)
set_seq2() (difflib.SequenceMatcher method)
set_seq() (difflib.SequenceMatcher method)
set_seqs() (difflib.SequenceMatcher method)
set_sequences() (mailbox.MH method)
        (mailbox.MHMessage method)
set_server_documentation() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
        (xmlrpc.server.DocXMLRPCServer method)
set_server_name() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
        (xmlrpc.server.DocXMLRPCServer method)
set_server_title() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
        (xmlrpc.server.DocXMLRPCServer method)
set_servername_callback (ssl.SSLContext attribute)
set_spacing() (formatter.formatter method)
set_start_method() (in module multiprocessing)
set_startup_hook() (in module readline)
set_step() (bdb.Bdb method)
set_subdir() (mailbox.MaildirMessage method)
set_task_factory() (asyncio.loop method)
set_terminator() (asynchat.async_chat method)
set_threshold() (in module gc)
set_trace() (bdb.Bdb method)
        (in module bdb)
        (in module pdb)
        (pdb.Pdb method)
set_trace_callback() (sqlite3.Connection method)
set_tunnel() (http.client.HTTPConnection method)
set_type() (email.message.Message method)
set_unittest_reportflags() (in module doctest)
set_unixfrom() (email.message.EmailMessage method)
set_until() (bdb.Bdb method)
set_url() (urllib.robotparser.RobotFileParser method)
set_usage() (optparse.OptionParser method)
set_userptr() (curses.panel.Panel method)
set_visible() (mailbox.BabylMessage method)
set_wakeup_fd() (in module signal)
set_write_buffer_limits() (asyncio.WriteTransport method)
setacl() (imaplib.IMAP4 method)
setannotation() (imaplib.IMAP4 method)
setattr() (built-in function)
setAttribute() (xml.dom.Element method)
setAttributeNode() (xml.dom.Element method)
setAttributeNodeNS() (xml.dom.Element method)
SetBase() (xml.parsers.expat.xmlparser method)
setblocking() (socket.socket method)
setByteStream() (xml.sax.xmlreader.InputSource method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
(wave.Wave_write method)
setContentHandler()
(xml.sax.xmlreader.XMLReader method)
setcontext() (in module decimal)
setDaemon() (threading.Thread method)
setdefault() (dict method)
(http.cookies.Morsel method)
setdefaulttimeout() (in module socket)
setdlopenflags() (in module sys)
setDocumentLocator()
(xml.sax.handler.ContentHandler method)
setDTDHandler()
(xml.sax.xmlreader.XMLReader method)
setegid() (in module os)
setEncoding() (xml.sax.xmlreader.InputSource method)
setEntityResolver()
(xml.sax.xmlreader.XMLReader method)
seteuid() (in module os)
setFeature() (xml.sax.xmlreader.XMLReader method)
setfirstweekday() (in module calendar)
setfmt() (ossaudiodev.oss_audio_device method)
setFormatter() (logging.Handler method)
(setau.AU_write method)
setgid() (in module os)
setframerate() (aifc.aifc method)
(sunau.AU_write method)
(wave.Wave_write method)
seth() (in module turtle)
setheading() (in module turtle)
sethostname() (in module socket)
SetInteger() (msilib.Record method)
setItem() (in module operator)
setitimer() (in module signal)
setLevel() (logging.Handler method)
(setting.Logger method)
setlocale() (in module locale)
setLocale() (xml.sax.xmlreader.XMLReader method)
setLoggerClass() (in module logging)
setlogmask() (in module syslog)
setLogRecordFactory() (in module logging)
setmark() (aifc.aifc method)
setMaxConns()
setmode() (in module msvcrt)
setName() (threading.Thread method)
setnchannels() (aifc.aifc method)
(setau.AU_write method)
(setwave.Wave_write method)
setnframes() (aifc.aifc method)
(setau.AU_write method)
(setwave.Wave_write method)
SetParamEntityParsing()
(xml.parsers.expat.xmlparser method)
setparameters() (ossaudiodev.oss_audio_device method)
setparams() (aifc.aifc method)
(setau.AU_write method)
(setwave.Wave_write method)
SetPassword() (zipfile.ZipFile method)
setpgid() (in module os)
setpgrp() (in module os)
setpos() (aifc.aifc method)
(in module turtle)
(sunau.AU_read method)
(wave.Wave_read method)
setposition() (in module turtle)
setpriority() (in module os)
setprofile() (in module sys)
(set in module threading)
SetProperty() (msilib.SummaryInformation method)
setProperty() (xml.sax.xmlreader.XMLReader method)
setPublicId() (xml.sax.xmlreader.InputSource method)
setquota() (imaplib.IMAP4 method)
setraw() (in module tty)
setrecursionlimit() (in module sys)
setregid() (in module os)
setresgid() (in module os)
setresuid() (in module os)
setreuid() (in module os)
setrlimit() (in module resource)
setsampwidth() (aifc.aifc method)
(sunau.AU_write method)
(wave.Wave_write method)
setscrreg() (curses.window method)
setsid() (in module os)
setsockopt() (socket.socket method), [1], [2]
setState() (codecs.IncrementalDecoder method)
(codecs.IncrementalEncoder method)
(in module random)
setStream() (logging.StreamHandler method)
SetStream() (msilib.Record method)
SetString() (msilib.Record method)
setswitchinterval() (in module sys), [1]
string_at() (in module string)
StringIO (class in io)
stringprep (in module strings, documentation)
strip() (bytearray method)
(bytes method)
(strip method)
strip_dirs() (pickle module)
strip_python_strerr() (in module test.support)
stripspaces() (re.Pattern method)
strptime() (datetime.datetime class method)
(strtobool() (in module distutils.util)
struct (module)
Struct (class in struct)
struct (module)
struct_sequence
structure
structures
C
strxfrm() (in module locale)
STType (in module parser)
styless
coding
Style (class in tkinter.ttk)
sub() (in module re)
(submethod)
(re.Pattern method)
attribute)
subclassing
immutable
(in module test.support)
setSystemId()  (xml.sax.xmlreader.InputSource method)
setsyx() (in module curses)
setTarget()  (logging.handlers.MemoryHandler method)
settiltangle() (in module turtle)
settimeout()  (socket.socket method)
setTimeOut()  (urllib.request.CacheFTPHandler method)
setup() (in module distutils.core)
  (socketserver.BaseRequestHandler method)
setUp() (unittest.TestCase method)
SETUP_ANNOTATIONS (opcode)
SETUP_ASYNC_WITH (opcode)
setup_environ()  (wsgiref.handlers.BaseHandler method)
SETUP_EXCEPT (opcode)
SETUP_FINALLY (opcode)
SETUP_LOOP (opcode)
setup_python()  (venv.EnvBuilder method)
setup_scripts()  (venv.EnvBuilder method)
setup_testing_defaults() (in module wsgiref.util)
SETUP_WITH (opcode)
setUpClass() (unittest.TestCase method)
setUpTerm() (in module curses)
SetValue() (in module winreg)
SetValueEx() (in module winreg)
shortDescription() (unittest.TestCase method)  SupportsBytes
shorten() (in module textwrap)  SupportsComplex
shouldFlush()  SupportsFloat
 logging.handlers.BufferingHandler method)
     (logging.handlers.MemoryHandler method)
shouldStop (unittest.TestResult attribute)  SupportsInt
show() (curses.panel.Panel method)  SupportsRound
show_code() (in module dis)  SW_HIDE (in module subprocess)
show_compilers() (in module distutils.ccompiler)  swap_attr() (in module test.support)
showsyntaxerror()  (code.InteractiveInterpreter method)
showtraceback()  (code.InteractiveInterpreter method)
 showturtle() (in module turtle)  swapcase() (in module test.support)
showwarning() (in module warnings)  swap_item() (in module test.support)
shuffle() (in module random)  swap_attr()
shutdown()  (concurrent.futures.Executor method)
    (imaplib.IMAP4 method)
    (in module logging)
    (multiprocessing.managers.BaseManager method)
    (socket.socket method)
    (socketserver.BaseServer method)
shutdown_asyncgens() (asyncio.loop method)
shutil (module)  symlinking()
    (imaplib.IMAP4 method)
    (in module logging)
    (multiprocessing.managers.BaseManager method)
    (socket.socket method)
    (socketserver.BaseServer method)
    (imaplib.IMAP4 method)
    (in module logging)
    (multiprocessing.managers.BaseManager method)
    (socket.socket method)
    (socketserver.BaseServer method)
shutil_asyncgens() (asyncio.loop method)
shutil (module)  symlinking()
side_effect (unittest.mock.Mock attribute)  (in module multiprocessing.sharedctypes)
SIG_BLOCK (in module signal)  SyncManager
SIG_DFL (in module signal)  SyncManager
SIG_IGN (in module signal)  SyncManager
SIG_SETMASK (in module signal)  synchronized()
SIG_UNBLOCK (in module signal)  synchronized()
SIGINT, [1]  SyncManager
siginterrupt() (in module signal)  SyncManager
signal
  module, [1]
signal (module)
signal() (in module signal)
Signature (class in inspect)
signature (inspect.BoundArguments attribute)
signature() (in module inspect)
sigpending() (in module signal)
sigtimedwait() (in module signal)
sigwait() (in module signal)
sigwaitinfo() (in module signal)
simple
  statement
Simple Mail Transfer Protocol
SimpleCookie (class in http.cookies)
simplefilter() (in module warnings)
SimpleHandler (class in wsgiref.handlers)
syncok() (curses.window method)
syncup() (curses.window method)
syntax
SyntaxErr
SyntaxError
SyntaxWarning
sys
  module,
sys (module)
sys.exc_infosys.last_tracebacksys.meta_pathsys.modulessys.pathsys.path_hookssys.path_importer_cachesys.stderrsys.stdinsys.stdout
sys_exc (2to3 fixer)
sys_version (http.server.BaseHTTPRequestHandler attribute)
sysconf() (in
sysconf_names (in
sysconfig (module)
syslog (module)
syslog() (in
SysLogHandler (class in
system() (in
  (in module
system_alias()
system_must_validate_cert() (in
test.support)
Index – T

T_FMT (in module locale)
T_FMT_AMPM (in module locale)
tab
tab() (tkinter.ttk.Notebook method)
TabError
tabnanny (module)
tabs() (tkinter.ttk.Notebook method)
tabsize  (textwrap.TextWrapper attribute)
tabular
data
tag  (xml.etree.ElementTree.Element attribute)
tag_bind()  (tkinter.ttk.Treeview method)
tag_configure()  (tkinter.ttk.Treeview method)
tag_has()  (tkinter.ttk.Treeview method)
tagName (xml.dom.Element attribute)
tail  (xml.etree.ElementTree.Element attribute)
take_snapshot()  (in module tracemalloc)
takewhile()  (in module itertools)
tan()  (in module cmath)
        (in module math)
tanh()  (in module cmath)
        (in module math)
TarError
TarFile (class in tarfile), [1]
times()  (in module os)
tIMESTAMP
(py_compile.PycInvalidationMode attribute)
timestamp()  (datetime.datetime method)
timetuple()  (datetime.datetime method)
timetz()  (datetime.datetime method)
timezone (class in datetime)
        (in module time)
title()  (bytestring method)
        (bytestring method)
        (in module turtle)
        (str method)
Tix
tix_addbitmapdir()  (tkinter.tix.tixCommand method)
tix_cget()  (tkinter.tix.tixCommand method)
tix_configure()  (tkinter.tix.tixCommand method)
tix_filedialog()  (tkinter.tix.tixCommand method)
tix_getbitmap()  (tkinter.tix.tixCommand method)
tix_getimage()  (tkinter.tix.tixCommand method)
tix_option_get()  (tkinter.tix.tixCommand method)
tarfile (module)
tarfile command line option
   --create  <tarfile>  <source1> ...  <sourceN>
   --extract <tarfile> [<output_dir>]
   --list <tarfile>
   --test <tarfile>
   -c  <tarfile>  <source1> ...  <sourceN>
   -e <tarfile> [<output_dir>]
   -l <tarfile>
   -t <tarfile>
   -v, --verbose
target
deletion
   list, [1]
   list assignment
   list, deletion
   loop control
target
   (xml.dom.ProcessingInstruction attribute)
TarInfo (class in tarfile)
Task (class in asyncio)
task_done() (asyncio.Queue method)
   (multiprocessing.JoinableQueue method)
   (queue.Queue method)
tau (in module cmath)
   (in module math)
tb_frame (traceback attribute)
tb_lasti (traceback attribute)
tb_lineno (traceback attribute)
tb_locals (unittest.TestResult
   (tix_resetoptions() (tkinter.tix.tixCommand
   (class in tkinter)
   (class in tkinter.tix)
Tk Option Data Types
TK_LIBRARY
Tkinter
tkinter (module)
tkinter.scrolledtext (module)
tkinter.tix (module)
tkinter.ttk (module)
TList (class in tkinter.tix)
TLS
TLSv1 (ssl.TLSVersion
TLSv1_1 (ssl.TLSVersi
TLSv1_2 (ssl.TLSVersi
TLSv1_3 (ssl.TLSVersi
TLSVersion (class in ss
TMP
TMPDIR
to_bytes() (int method)
to_eng_string() (decimal
   (decimal.Decimal r
to_integral() (decimal
   to_integral_exact() (decimal
   to_integral_value() (decimal
   to_sci_string() (decimal
attribute)
tb_next (traceback attribute)
tbreak (pdb command)
tcdrain() (in module termios)
tcflow() (in module termios)
tcflush() (in module termios)
tcgetattr() (in module termios)
tcgetattrs() (in module termios)
tc_gettime() (in module os)
Tcl() (in module tkinter)
TCL_LIBRARY
TCPserver (class in socketserver)
tc_sendbreak() (in module termios)
tc_setattrs() (in module termios)
tc_gettime() (in module os)
tearDown() (unittest.TestCase method)
tearDownClass() (unittest.TestCase method)
tee() (in module itertools)
tell() (aifc.aifc method), [1]
(chunk.Chunk method)
(io.IOBase method)
(io.TextIOBase method)
(mmap.mmap method)
(suna.AU_read method)
(suna.AU_write method)
(wave.Wave_read method)
(wave.Wave_write method)
Telnet (class in telnetlib)
telnetlib (module)
TEMP
temp_cwd() (in module test.support)
temp_dir() (in module test.support)
temp_umask() (in module os)
ToASCII() (in encodings.idna)
tobuf() (tarfile.TarInfo method)
tobytes() (array.array method)
tomorrow() (datetime.datetime method)
today() (datetime.datetime method)
toifile() (array.array method)
tok_name (in module tokenize)
token
(token)
(token_bytes) (in module secrets)
token_hex() (in module secrets)
token_urlsafe() (in module secrets)
TokenError
tokenize (module)
tokenize command line
-e, --exact
-h, --help
tokenize() (in module tokenize)
tolist() (array.array method)
(memoryview method)
(parser.ST method)
tomono() (in module audioop)
toordinal() (datetime.datetime method)
top() (curses.panel.Panel method)
(top_panel) (in poplib.POP3 method)
top_panel() (in curses.panel)
membership
test (doctest.DocTestFailure attribute)
  (doctest.UnexpectedException attribute)
  (module)
test() (in module cgi)
test.support (module)
test.support.script_helper (module)
TEST_DATA_DIR (in module test.support)
TEST_HOME_DIR (in module test.support)
TEST_HTTP_URL (in module test.support)
TEST_SUPPORT_DIR (in module test.support)
TestCase (class in unittest)
TestFailed
testfile() (in module doctest)
TESTFN (in module test.support)
TESTFN_ENCODING (in module test.support)
TESTFN_NONASCII (in module test.support)
TESTFN_UNDECODABLE (in module test.support)
TESTFN_UNENCODABLE (in module test.support)
TESTFN_UNICODE (in module test.support)
TestHandler (class in test.support)
TestLoader (class in unittest)
testMethodPrefix (unittest.TestLoader attribute)
trace (module)
  trace command line opt
    -help
    --ignore-dir=<dir>
    --ignore-module=<mod>
    --version
    -C, --coverdir=<dir>
    -R, --no-report
    -T, --trackcalls
    -c, --count
    -f, --file=<file>
    -g, --timing
    -l, --listfuncs
    -m, --missing
    -r, --report
    -s, --summary
    -t, --trace
  trace function, [1], [2]
trace() (in module inspect)
trace_dispatch() (bdb.Bdb method)
Traceback (class in tracemalloc)
  traceback (module)
    (tracemalloc.Statistic attribute)
    (tracemalloc.StatisticDiff attribute)
    (tracemalloc.Trace attribute)
  traceback_limit
  (tracemalloc.Snapshot attribute)
  (wsgiref.handlers.Exception attribute)
TracebackException
testmod() (in module doctest)
testNamePatterns
    (unittest.TestLoader attribute)
TestResult (class in unittest)
tests (in module imghdr)
testsource() (in module doctest)
testsRun (unittest.TestResult attribute)
TestSuite (class in unittest)
testzip() (zipfile.ZipFile method)
Text (class in typing)
text (in module msilib)
    (traceback.TracebackException attribute)
    (xml.etree.ElementTree.Element attribute)
text encoding
  text file
text mode
text() (in module cgitb)
    (msilib.Dialog method)
text_factory (sqlite3.Connection attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
    (in module locale)
TextFile (class in distutils.text_file)
textinput() (in module turtle)
TextIO (class in typing)
TextIOBase (class in io)
TextIOWrapper (class in io)
TextTestResult (class in unittest)
TextTestRunner (class in unittest)
traceback)
tracebacklimit (in module traceback)
    (in CGI scripts)
TracebackType (class in traceback)
tracer() (in module turtle)
traces (traceback arousing)
trailing comma
transfercmd() (ftpLib.FTIl)
transient_internet() (in module test.support)
TransientResource
test.support)
translate() (bytearray method)
    (in module fnmatch)
    (str method)
translation() (in module
transport (asyncio.StreamWriter
Transport (class in asyncio)
Transport Layer Security
traverseproc (C type)
Tree (class in tkinter.tix)
TreeBuilder (class
xml.etree.ElementTree)
Treeview (class in tkinter)
triangular() (in module random)
true, [1], [2]
textwrap (module)
TextWrapper (class in textwrap)
theme_create() (tkinter.ttk.Style method)
theme_names() (tkinter.ttk.Style method)
theme_settings() (tkinter.ttk.Style method)
theme_use() (tkinter.ttk.Style method)
THOUSEP (in module locale)
Thread (class in threading)
thread() (imaplib.IMAP4 method)
thread_info (in module sys)
thread_time() (in module time)
thread_time_ns() (in module time)
threading (module)
threading_cleanup() (in module test.support)
threading_setup() (in module test.support)
ThreadingHTTPServer (class in http.server)
ThreadingMixIn (class in socketserver)
ThreadingTCPServer (class in socketserver)
ThreadingUDPServer (class in socketserver)
ThreadPoolExecutor (class in concurrent.futures)
threads
    POSIX
throw (2to3 fixer)
throw() (coroutine method)

True (built-in variable)
truediv() (in module operator)
trunc() (in module math)
truncate() (in module os)
(io.IOBase method)
truth
  value
truth() (in module operator)
try
  statement, [1], [2]
ttk
tty
   I/O control
tty (module)
ttyname() (in module os)
tuple
  built-in function, [1]
display
  empty, [1]
object, [1], [2], [3], singleton
tuple (built-in class)
Tuple (in module typing)
tuple2st() (in module parser)
turtle_params (2to3 fixer)
Turtle (class in turtle)
turtle (module)
turtledemo (module)
turtles() (in module turtle)
TurtleScreen (class in turtle)
turtlesize() (in module turtle)
type, [1]
    Boolean
    built-in function, [1]
(generator method)
ticket_lifetime_hint  (ssl.SSLSession attribute)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
TILDE (in module token)
tilt() (in module turtle)
tiltangle() (in module turtle)
time (class in datetime)
  (module)
  (ssl.SSLSession attribute)
time() (asyncio.loop method)
  (datetime.datetime method)
  (in module time)
Time2Internaldate() (in module imaplib)
time_ns() (in module time)
timedelta (class in datetime)
TimedRotatingFileHandler (class in logging.handlers)
timegm() (in module calendar)
timeit (module)
timeit command line option
  -h, --help
  -n N, --number=N
  -p, --process
  -r N, --repeat=N
  -s S, --setup=S
  -u, --unit=U
  -v, --verbose
timeit() (in module timeit)
  (timeit.Timer method)
timeout  

(data
  hierarchy
  immutable data
  object, [1], [2]
  operations on dict
  operations on list
  type (built-in class)
  Type (class in typing)
  type (optparse.Option attribute)
  (socket.socket attribute)
  (tarfile.TarInfo attribute)
  (urllib.request.Request attribute)

type alias

type hint
  type of an object
  TYPE_CHECKER
  (optparse.Option attribute)
  TYPE_CHECKING (in typing)
typeahead() (in module
  typecode (array.array attribute)
typecodes (in module a
  TYPED_ACTIONS
  (optparse.Option attribute)
typed_subpart_iterator( email.iterators)
  TypeError
    exception

types
  built-in
  immutable sequence
  mutable sequence
timeout() (curses.window method)
TIMEOUT_MAX (in module _thread)
TimeoutError, [1], [2], [3]
TimeoutExpired
Timer (class in threading)
    (class in timeit)
TimerHandle (class in asyncio)
Index – U

u"
   string literal
u'
   string literal
u-LAW, [1], [2]
ucd_3_2_0 (in module unicodedata)
udata (select.kevent attribute)
UDPServer (class in socketserver)
UF_APPEND (in module stat)
UF_COMPRESSED (in module stat)
UF_HIDDEN (in module stat)
UF_IMMUTABLE (in module stat)
UF_NODUMP (in module stat)
UF_NOUNLINK (in module stat)
UF_OPAQUE (in module stat)
uid (tarfile.TarInfo attribute)
uid() (imaplib.IMAP4 method)
uidl() (poplib.POP3 method)
ulaw2lin() (in module audioop)
ULONG_MAX
umask() (in module os)
unalias (pdb command)
uname (tarfile.TarInfo attribute)
uname() (in module os)
   (in module platform)
unary
   arithmetic operation
   bitwise operation
UNARY_INVERT (opcode)
UNARY_NEGATIVE (opcode)
UNARY_NOT (opcode)
unquote_plus() (in module urllib.parse)
unquote_to_bytes()
unreadline() (distutils.text_file method)
unrecognized escape sequence
unregister() (in module atexit)
unregister_archive_format()
unregister_dialect()
unregister_dialect()
unsafe (uuid.SafeUUID attribute)
unset() (test.support.EnvironmentVarGuard method)
unsetenv() (in module os)
UnstructuredHeader (class in email.headerregistry)
unsubscribe() (imaplib.IMAP4 method)
UnsupportedOperation
until (pdb command)
untokenize() (in module tokenize)
untouchwin() ( curses.window method)
unused_data (bz2.BZ2Decompressor attribute)
   (lzma.LZMADecompressor attribute)
   (zlib.Decompress attribute)
unverifiable (urllib.request.Request attribute)
unwrap() (in module inspect)
   (ssl.SSLSocket method)
UNARY_POSITIVE (opcode)
unbinding
    name
UnboundLocalError, [1]
unbuffered I/O
UNC paths
    and os.makedirs()
UNCHECKED_HASH
    (py_compile.PycInvalidationMode attribute)
unconsumed_tail
    (zlib.Decompress attribute)
unctrl() (in module curses)
    (in module curses.ascii)
undefine_macro()
    (distutils.ccompiler.CCompiler method)
Underflow (class in decimal)
undisplay (pdb command)
undo() (in module turtle)
undobufferentries() (in module turtle)
undoc_header (cmd.Cmd attribute)
unescape() (in module html)
    (in module xml.sax.saxutils)
UnexpectedException
unexpectedSuccesses (unittest.TestResult attribute)
unfreeze() (in module gc)
unget_wch() (in module curses)
    (in module msvcr)
ungetmouse() (in module curses)
ungetwch() (in module msvcr)
unhexlify() (in module binascii)
Unicode, [1], [2]
    database
unicode (2to3 fixer)
Unicode Consortium
unicodedata (module)
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeWarning
unidata_version (in module unicodedata)
unified_diff() (in module difflib)
uniform() (in module random)
UnimplementedFileMode
Union (class in ctypes)
    (in module typing)
union() (frozenset method)
unique() (in module enum), [1]
unittest (module)

unittest command line option
    --locals
    -b, --buffer
    -c, --catch
    -f, --failfast
    -k

unittest-discover command line option
    -p, --pattern pattern
    -s, --start-directory directory
    -t, --top-level-directory directory
    -v, --verbose

unittest.mock (module)

universal newlines
    What's new, [1], [2], [3]
    bytearray.splitlines method
    bytes.splitlines method
    csv.reader function

urllib (2to3 fixer)
    (module)
urllib.error (module)
urllib.parse (module)
urllib.request (module)
urllib.response (module)
urllib.robotparser (module)
urlopen() (in module)
URLopener (class)
urllparse() (in module)
urlretrieve() (in module)
urlsafe_b64decode() (in module)
urlsafe_b64encode() (in module)
urlsplit() (in module)
urlunparse() (in module)
urlunsplit() (in modules)
urn (uuid.UUID attribute)
use_default_colors() (in module)
use_env() (in module)
use_rawinput (cmd attribute)
UseForeignDTD(method)
USER
    user
effective id
    id
    id, setting
user() (poplib.POP3 method)
user-defined
    function
    function call
    method
unpack() (in module struct) (struct.Struct method)
unpack_archive() (in module shutil)
unpack_array() (xdrlib.Unpacker method)
unpack_bytes() (xdrlib.Unpacker method)
unpack_double() (xdrlib.Unpacker method)
UNPACK_EX (opcode)
unpack_farray() (xdrlib.Unpacker method)
unpack_float() (xdrlib.Unpacker method)
unpack_fopaque() (xdrlib.Unpacker method)
unpack_from() (in module struct) (struct.Struct method)
unpack_fstring() (xdrlib.Unpacker method)
unpack_list() (xdrlib.Unpacker method)
unpackOpaque() (xdrlib.Unpacker method)
UNPACK_SEQUENCE (opcode)
unpack_string() (xdrlib.Unpacker method)
Unpacker (class in xdrlib)

unpacking
dictionary
in function calls
iterable

unparsedEntityDecl()
(xml.sax.handler.DTDHandler method)
UnparsedEntityDeclHandler()
(xml.parsers.expat.xmlparser method)
Unpickler (class in pickle)
UnpicklingError
unquote() (in module email.utils)
(in module urllib.parse)
Index – V

v4_int_to_packed() (in module ipaddress)
v6_int_to_packed() (in module ipaddress)
validator() (in module wsgiref.validate)
value
default parameter
truth
value (ctypes._SimpleCData attribute)
  (http.cookiejar.Cookie attribute)
  (http.cookies.Morsel attribute)
  (xml.dom.Attr attribute)
value of an object
Value() (in module multiprocessing)
  (in module multiprocessing.sharedctypes)
  (multiprocessing.managers.SyncManager method)
value_decode() (http.cookies.BaseCookie method)
value_encode() (http.cookies.BaseCookie method)
ValueError
  exception
valuerefs() (weakref.WeakValueDictionary method)
values
  Boolean
  writing
values() (contextvars.Context method)
  (dict method)
  (email.message.EmailMessage method)
  (email.message.Message method)
  (mailbox.Mailbox method)
verify_client_post_handshake()
verify_code (ssl.SSLCertVerificationError attribute)
VERIFY_CRL_CHECK_CHAIN (in module ssl)
VERIFY_CRL_CHECK_LEAF (in module ssl)
VERIFY_DEFAULT (in module ssl)
verify_flags (ssl.SSLCertVerificationError attribute)
verify_mode (ssl.SSLContext attribute)
verify_request()
VERIFY_X509 (ssl.SSLContext attribute)
VERIFY_X509_TRUSTED_FIRST (in module ssl)
VerifyFlags (class in ssl)
VerifyMode (class in ssl)
version
valuerefs() (weakref.WeakValueDictionary method)
values
  (http.client.HTTPResponse attribute)
  (http.cookiejar.Cookie attribute)
  (in module curses)
  (in module marshal)
  (in module sqlite3)
  (in module sys)
  (ipaddress.IPv4Address attribute)
  (ipaddress.IPv4Network attribute)
Index – W

W_OK (in module os)
wait()
(asyncio.asyncio.subprocess.Process method)
   (asyncio.Condition method)
   (asyncio.Event method)
   (in module asyncio)
   (in module concurrent.futures)
   (in module multiprocessing.connection)
   (in module os)
   (multiprocessing.poolAsyncResult method)
   (subprocess.Popen method)
   (threading.Barrier method)
   (threading.Condition method)
   (threading.Event method)
wait3() (in module os)
wait4() (in module os)
wait_closed() (asyncio.Server method)
   (asyncio.StreamWriter method)
wait_for() (asyncio.Condition method)
   (in module asyncio)
   (threading.Condition method)
wait_threads_exit() (in module test.support)
waitid() (in module os)
waitpid() (in module os)
walk() (email.message.EmailMessage method)
   (email.message.Message method)
walk_packages() (in module pkgutil)
walk_stack() (in module traceback)
walk_tb() (in module traceback)
want (doctest.Example attribute)
warn() (distutils.ccompiler.CCompiler method)
            (distutils.text_file.TextFile method)
            (in module warnings)
warn_explicit() (in module warnings)
Warning, [1]
warning() (in module logging)
            (logging.Logger method)
            (xml.sax.handler.ErrorHandler method)
warnings
            (module)
WarningsRecorder (class in test.support)
warnoptions (in module sys)
wasSuccessful() (unittest.TestResult method)
WatchedFileHandler (class in logging.handlers)
wave (module)
WCONTINUED (in module os)
WCOREDUMP() (in module os)
WeakKeyDictionary (class in weakref)
WeakMethod (class in weakref)
weakref (module)
WeakSet (class in weakref)
WeakValueDictionary (class in weakref)
wrap_future() (in module asyncio)
wrap_socket() (in module ssl)
wrap_text() (distutils.fancy_getopt)
wrapper() (in module WeakMethod)
WrapperDescriptorType (in module types)
wraps() (in module functools)
WRITABLE (in module xml.etree.ElementTree)
writable() (asyncore.dispatcher method)
write() (asyncio.StreamWriter method)
            (asyncio.WriteTransport method)
            (code.InteractiveInterpreter method)
            (codecs.StreamWriter method)
            (configparser.ConfigParser method)
            (email.generator.Message method)
            (email.generator.MessageParts method)
            (in module turtle)
            (in module os)
            (io.BufferedIOBase method)
            (io.BufferedWriter method)
            (io.RawIOBase method)
            (mmap.mmap method)
            (ossaudiodev.oss_audio_device method)
            (ssl.MemoryBIO method)
            (ssl.SSLContext method)
            (telnetlib.Telnet method)
            (xml.etree.ElementTree method)
webbrowser (module)

weekday() (datetime.date method)
  (datetime.datetime method)
  (in module calendar)

weekheader() (in module calendar)

weibullvariate() (in module random)

WEXITED (in module os)

WEXITSTATUS() (in module os)

wfile
  (http.server.BaseHTTPRequestHandler attribute)

what() (in module imghdr)
  (in module sndhdr)

whathdr() (in module sndhdr)

whatis (pdb command)

when() (asyncio.TimerHandle method)

where (pdb command)

which() (in module shutil)

whichdb() (in module dbm)

while
  statement, [1], [2], [3]

whitespace (in module string)
  (shlex.shlex attribute)

whitespace_split (shlex.shlex attribute)

Widget (class in tkinter.ttk)

width (textwrap.TextWrapper attribute)

width() (in module turtle)

WIFCONTINUED() (in module os)

WIFEXITED() (in module os)

WIFSIGNALED() (in module os)

WIFSTOPPED() (in module os)

win32_ver() (in module platform)

WinDLL (class in ctypes)

window manager (widgets)

write_byte() (mmap.mmap method)

write_bytes() (pathlib.Path method)

write_docstringdict() (turtle method)

write_eof() (asyncio StreamWriter method)

write_file() (in module write_file)

write_history_file() (in asyncio.WriteTransport method)

write_results() (traceback method)

write_text() (pathlib.Path method)

write_through (io.TextIOWrapper attribute)

writeall() (ossaudiodev.oss_audio_device method)

writeframes() (aifc.aifc method)
  (sunau.AU_write method)

writeframesraw() (aifc method)
  (sunau.AU_write method)

writeheader() (csv.DictWriter method)

writelines() (asyncioStreamWriter method)

writePlist() (in module writePlistToBytes() (in module writePlistToBytes() (in module writepy() (zipfile.PyZipFile method)

writer (formatter.formatter attribute)
window() (curses.panel.Panel method)
window_height() (in module turtle)
window_width() (in module turtle)
Windows
Windows ini file
WindowsError
WindowsPath (class in pathlib)
WindowsProactorEventLoopPolicy (class in asyncio)
WindowsRegistryFinder (class in importlib.machinery)
winerror (OSError attribute)
WinError() (in module ctypes)
WINFUNCTYPE() (in module ctypes)
winreg (module)
WinSock
winsound (module)
winver (in module sys)
writer() (in module csv)
writerow() (csv.csvwriter method)
writerows() (csv.csvwriter method)
writestr() (zipfile.ZipFile method)
WriteTransport (class in asyncio)
writev() (in module os)
writexml() (xml.dom.minidom.Node method)
writing values
WrongDocumentErr
ws_comma (2to3 fixer)
wsgi_file_wrapper (wsgiref.handlers.Base)
wsgi_multiprocess (wsgiref.handlers.Base)
wsgi_multithread (wsgiref.handlers.Base)
wsgi_run_once (wsgiref.handlers.Base)
wsgiref (module)
wsgiref.handlers (module)
wsgiref.headers (module)
wsgiref.simple_server (module)
wsgiref.util (module)
wsgiref.validate (module)
WSGIRequestHandler (wsgiref.simple_server)
WSGIServer (wsgiref.simple_server)
wShowWindow (subprocess.STARTUPINFO attribute)
WSTOPPED (in module os)
WSTOPSIG() (in module os)
wstring_at() (in module ctypes)
WTERMSIG() (in module os)
WUNTRACED (in module os)

WWW, [1], [2]
server, [1]
Index – X

X (in module re)
X509 certificate
X_OK (in module os)
xatom() (imaplib.IMAP4 method)
XATTR_CREATE (in module os)
XATTR_REPLACE (in module os)
XATTR_SIZE_MAX (in module os)
xcor() (in module turtle)
XDR
xdrlib (module)
xhdr() (nntplib.NNTP method)
XHTML
XHTML_NAMESPACE (in module xml.dom)
xml (module)
XML() (in module xml.etree.ElementTree)
xml.dom (module)
xml.dom.minidom (module)
xml.dom.pulldom (module)
xmll.etree.ElementTree (module)
xmll.parsers.expat (module)
xmll.parsers.expat.errors (module)
xmll.parsers.expat.model (module)
xmll.sax (module)
xmll.sax.handler (module)
xmll.sax.saxutils (module)
xmll.sax.xmlreader (module)
XML_ERROR_ABORTED (in module xml.parsers.expat.errors)
XML_ERROR_ASYNC_ENTITY (in module xml.parsers.expat.errors)
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
module xml.parsers.expat.errors)
XML_ERROR_BAD_CHAR_REF   (in   module
xml.parsers.expat.errors)
XML_ERROR_BINARY_ENTITY_REF (in   module
xml.parsers.expat.errors)
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (in module xml.parsers.expat.errors)
XML_ERROR_DUPLICATE_ATTRIBUTE (in   module
xml.parsers.expat.errors)
XML_ERROR_ENTITY_DECLARED_IN_PE (in   module
xml.parsers.expat.errors)
XML_ERROR_EXTERNAL_ENTITY_HANDLING (in   module
xml.parsers.expat.errors)
XML_ERROR_FEATURE_REQUIRES_XML_DTD (in   module
xml.parsers.expat.errors)
XML_ERROR_FINISHED (in module xml.parsers.expat.errors)
XML_ERROR_INCOMPLETE_PE (in module xml.parsers.expat.errors)
XML_ERROR_INCORRECT_ENCODING (in module xml.parsers.expat.errors)
XML_ERROR_INCOMPLETE_PE (in module xml.parsers.expat.errors)
XML_ERROR_INCOMPLETE_PE (in module xml.parsers.expat.errors)
XML_ERROR_INVALID_TOKEN (in module xml.parsers.expat.errors)
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (in module xml.parsers.expat.errors)
XML_ERROR_MISPLACED_XML_PI (in module xml.parsers.expat.errors)
## Index – Y

<table>
<thead>
<tr>
<th>Y2K</th>
<th>ycor() (in module turtle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>year (datetime.date attribute)</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>(datetime.datetime attribute)</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>Year 2000</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>Year 2038</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>yeardatescalendar() (calendar.Calendar method)</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>yeardays2calendar() (calendar.Calendar method)</td>
<td>yeardays2calendar() (calendar.Calendar method)</td>
</tr>
<tr>
<td>YESEXPR (in module locale)</td>
<td>yield examples</td>
</tr>
<tr>
<td>yield expression</td>
<td>keyword statement</td>
</tr>
<tr>
<td>yield from (in What's New)</td>
<td>YIELD_FROM (opcode)</td>
</tr>
<tr>
<td>YIELD_VALUE (opcode)</td>
<td>yiq_to_rgb() (in module colorsys)</td>
</tr>
<tr>
<td>yview() (tkinter.ttk.Treeview method)</td>
<td>yview() (tkinter.ttk.Treeview method)</td>
</tr>
</tbody>
</table>
Zen of Python
ZeroDivisionError exception
zf1l() (bytearray method)
(bytestring method)
(str method)
zip (2to3 fixer)
zip() (built-in function)
ZIP_BZIP2 (in module zipfile)
ZIP_DEFLATED (in module zipfile)
zip_longest() (in module itertools)
ZIP_LZMA (in module zipfile)
ZIP_STORED (in module zipfile)
zipapp (module)

zipapp command line option
--info
-c, --compress
-h, --help
-m <mainfn>, --main=<mainfn>
-o <output>, --output=<output>
-p <interpreter>, --python=<interpreter>

ZipFile (class in zipfile)
zipfile (module)
zipfile command line option
--create <zipfile> <source1> ...
--extract <zipfile>
-output_dir
--list <zipfile>
--test <zipfile>
-c <zipfile> <source1> ...
-e <zipfile> <output_dir>
-l <zipfile>
-t <zipfile>

zipimport (module)
zipimporter (class in zipimport)
ZipImportError
ZipInfo (class in zipfile)
zlib (module)
ZLIB_RUNTIME_VERSION (in module zlib)
Index

| Symbols | _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | R | S | T | U | V | W | X | Y | Z |
Symbols

! (exclamation)
   in a command interpreter
   in curses module
   in formatted string literal
   in glob-style wildcards, [1]
   in string formatting
   in struct format strings

! (pdb command)
!=
   operator, [1]

" (double quote)
   string literal

......
   string literal

# (hash)
   comment, [1], [2]
   in doctests
   in printf-style formatting, [1]
   in regular expressions
   in string formatting
   source encoding declaration

$ (dollar)
   environment variables expansion
   in regular expressions
   in template strings
   interpolation in configuration files

% (percent)
   datetime format, [1], [2]

-O command line option
-o <output>, --output=

-OO command line option

-OO command line option

-p <interpreter>, --python=<interpreter>
   zipapp command line option

-p, --pattern pattern
   unittest-discover command line option

-p, --preamble=<preamble>
   pickletools command line option

-p, --process
   timeit command line option

-q command line option

-R command line option

-r compileall command
environment variables expansion (Windows), [1]
text interpolation in configuration files
operator, [1]
printf-style formatting, [1]
%=
  augmented assignment
& (ampersand)
  operator, [1]
&=
  augmented assignment
' (single quote)
  string literal
''
  string literal
() (parentheses)
  call
class definition
function definition
generator expression
in assignment target list
in printf-style formatting, [1]
in regular expressions
tuple display
(?)
in regular expressions
(?!)
in regular expressions
(?#)
in regular expressions
(?!)
in regular expressions
line option
-r N, --repeat=N
timeit command line option
-R, --no-report
trace command line option
-r, --report
trace command line option
-S
  command line option
-s
  command line option
-s S, --setup=S
timeit command line option
-s, --start-directory directory
unittest-discover command line option
-s, --summary
trace command line option
-t <tarfile>
tarfile command line option
-t < zipfile>
zipfile command line option
-t, --top-level-directory directory
unittest-discover command line option
-t, --trace
in regular expressions

in regular expressions

in regular expressions

in regular expressions

in regular expressions

* (asterisk)

function definition

import statement

in AST grammar

in argparse module

in assignment target list

in expression lists

in function calls, [1]

in glob-style wildcards, [1]

in printf-style formatting, [1]

in regular expressions

operator, [1]

**

function definition

in dictionary displays

in function calls, [1]

in glob-style wildcards

operator, [1]

**=

augmented assignment

*= augmented assignment

*?
in regular expressions
+ (plus)
  binary operator, [1]
in argparse module
in doctests
in printf-style formatting,
[1]
in regular expressions
in string formatting
unary operator, [1]
+=
  augmented assignment
+?
in regular expressions
, (comma)
  argument list
  expression list, [1], [2], [3],
  [4]
  identifier list, [1]
  import statement
in dictionary displays
in string formatting
in target list
parameter list
slicing
tuple display
with statement
-
(minus)
  binary operator, [1]
in doctests
in glob-style wildcards, [1]
in printf-style formatting,
[1]
in regular expressions
in numeric literal
in pathnames, [1]
in printf-style formatting,
[1]
in regular expressions
in string formatting
.
in pathnames
...
  ellipsis literal, [1], [2]
in doctests
  interpreter prompt, [1]
  placeholder, [1], [2]
.ini
  file
 .pdbrc
  file
 .pypirc file
/ (slash)
in pathnames, [1]
  operator, [1]
//
  operator, [1]
//=
  augmented assignment
/= augmented assignment
0b
  integer literal
0o
  integer literal
0x
  integer literal
2to3
in string formatting
unary operator, [1]
--check-hash-based-pycs
default|always|never
command line option
--create <tarfile> <source1> ... <sourceN>
tarfile command line option
--create <zipfile> <source1> ... <sourceN>
zipfile command line option
--details
inspect command line option
--extract <tarfile> [<output_dir>]
tarfile command line option
--extract <zipfile> <output_dir>
zipfile command line option
--help
command line option
trace command line option
--ignore-dir=<dir>
trace command line option
--ignore-module=<mod>
trace command line option
--info
zipapp command line option
--invalidation-mode [timestamp|checked-hash|unchecked-hash]
: (colon)
annotated variable
compound statement, [1], [2], [3], [4], [5], [6]
function annotations, [1]
in SQL statements
dictionary expressions
in formatted string literal
in string formatting
lambda expression
path separator (POSIX)
slicing
; (semicolon), [1]
< (less)
in string formatting
in struct format strings
operator, [1]
<<
operator, [1]
<<=
augmented assignment
<=
operator, [1]
<BLANKLINE>
= (equals)
assignment statement
class definition
function definition
in function calls
in string formatting
in struct format strings
==
operator, [1]
compileall command line option
--list <tarfile>
tarfile command line option
--list <zipfile>
zipfile command line option
--locals
unittest command line option
--sort-keys
json.tool command line option
--test <tarfile>
tarfile command line option
--test <zipfile>
zipfile command line option
--user-base
site command line option
--user-site
site command line option
--version
command line option
trace command line option
-= augmented assignment
-> function annotations, [1]
-? command line option
-a, --annotate
pickletools command line option

> (greater)
in string formatting
in struct format strings
operator, [1]
>= operator, [1]
>> operator, [1]
>>> augmented assignment
>>> interpreter prompt, [1]
? (question mark)
in AST grammar
in SQL statements
in a command
interpreter
in argparse module
in glob-style wildcards,
[1]
in regular expressions
in struct format strings,
[1]
replacement character
?? in regular expressions
@ (at)
class definition
function definition
in struct format strings
operator
[] (square brackets)
in assignment target list
in glob-style wildcards,
-B
  command line option
-b
  command line option
  compileall command line option
-b, --buffer
  unittest command line option
-c <command>
  command line option
-c <tarfile> <source1> ... <sourceN>
  tarfile command line option
-c <zipfile> <source1> ... <sourceN>
  zipfile command line option
-c, --catch
  unittest command line option
-c, --compress
  zipapp command line option
-c, --count
  trace command line option
-C, --coverdir=<dir>
  trace command line option
-d
  command line option
-d destdir
  compileall command line option
-E
[1] in regular expressions
  in string formatting
  list expression
  subscription
\ (backslash)
  escape sequence, [1]
  in pathnames
  in regular expressions,
  [1], [2]
\ escape sequence
  in regular expressions
\A in regular expressions
\a escape sequence
  in regular expressions
\B in regular expressions
\b escape sequence
  in regular expressions
\D in regular expressions
\d escape sequence
  in regular expressions
\f escape sequence
  in regular expressions
\g in regular expressions
command line option
-e <tarfile> [<output_dir>]
tarfile command line option
-e <zipfile> <output_dir>
zipfile command line option
-e, --exact
tokenize command line option
-f
compileall command line option
-f, --failfast
unittest command line option
-f, --file=<file>
trace command line option
-g, --timing
trace command line option
-h
command line option
-h, --help
json.tool command line option
timeit command line option
tokenize command line option
zipapp command line option
-I
command line option
-i
command line option
-i list

\N escape sequence, [1] in regular expressions

\n escape sequence in regular expressions

\r escape sequence in regular expressions

\S escape sequence in regular expressions

\s in regular expressions

\t escape sequence in regular expressions

\U escape sequence, [1] in regular expressions

\u escape sequence, [1] in regular expressions

\v escape sequence in regular expressions

\W in regular expressions

\w in regular expressions

\x escape sequence, [1] in regular expressions

\Z in regular expressions
compileall command line option
-J
command line option
-j N
compileall command line option
-k
unittest command line option
-l
compileall command line option
-l <tarfile>
tarfile command line option
-l <zipfile>
zipfile command line option
-l, --indentlevel=<num>
pickletools command line option
-l, --listfuncs
trace command line option
-m <mainfn>, --main=<mainfn>
zipapp command line option
-m <module-name>
command line option
-m, --memo
pickletools command line option
-m, --missing
trace command line option
-n N, --number=N

^ (caret)
in curses module
in regular expressions,
[1]
in string formatting
marker, [1]
operator, [1]

^=
  augmented assignment

{} (curly brackets)
dictionary expression
  in formatted string literal
  in regular expressions
  in string formatting
  set expression

| (vertical bar)
in regular expressions
  operator, [1]

|=  augmented assignment

~ (tilde)
home directory expansion
  operator, [1]
timeit command line option
_

_(underscore)_

gettext
in numeric literal, [1]
in string formatting
_, identifiers
__, identifiers
__abs__() (in module operator)
  (object method)
__add__() (in module operator)
  (object method)
__aiter__() (object method)
__all__
  (optional module attribute)
  (package variable)
__and__() (in module operator)
  (object method)
__anext__() (agen method)
  (object method)
__annotations__ (class attribute)
  (function attribute)
  (module attribute)
__await__() (object method)
__bases__ (class attribute), [1]
__bool__() (object method), [1]
__breakpointhook__ (in module sys)
__bytes__()
  (email.message.EmailMessage method)
  (email.message.Message method)

__kwdefaults__ (function attribute)
__le__() (in module operator)
  (instance method)
  (object method)
__len__() (email.message.EmailMessage method)
  (email.message.Message method)
  (mailbox.Mailbox method)
  (mapping object method)
  (object method)
__length_hint__() (object method)
__loader__
  (module attribute)
  (types.ModuleType attribute)
__lshift__() (in module operator)
  (object method)
__lt__() (in module operator)
  (instance method)
  (object method)
__main__
  module, [1], [2], [3],
  __main__ (module)
__matmul__() (in module operator)
  (object method)
__missing__()
  (collections.defaultdict method)
  (object method)
__mod__() (in module operator)
  (object method)
__module__ (class attribute)
  (function attribute)
  (method attribute)
__context__ (exception attribute)
  (traceback.TracebackException attribute)
__copy__() (copy protocol)
__debug__
  (built-in variable)
__deepcopy__() (copy protocol)
__defaults__ (function attribute)
__del__() (io.IOBase method)
  (object method)
__delattr__() (object method)
__delete__() (object method)
__delitem__() (email.message.EmailMessage method)
  (email.message.Message method)
  (in module operator)
  (mailbox.MH method)
  (mailbox.Mailbox method)
__dict__ (class attribute)
  (function attribute)
  (instance attribute)
  (module attribute), [1]
  (object attribute)
__dir__() (module attribute)
__dir__() (object method)
  (unittest.mock.Mock method)
__displayhook__ (in module sys)
__divmod__() (object method)
__doc__ (class attribute)
  (function attribute)
__radd__() (object method)
__rand__() (object method)
__rdivmod__() (object method)
__reduce__() (object method)
__reduce_ex__() (object method)
__repr__() (multiprocessing.managers method)
  (netrc.netrc method)
  (object method)
__reversed__() (object method)
__rfloordiv__() (object method)
__rlshift__() (object method)
__rmatmul__() (object method)
__rmod__() (object method)
__rmul__() (object method)
__ror__() (object method)
__round__() (fractions.Fraction method)
__rpow__() (object method)
__rrshift__() (object method)
__rshift__() (in module operator)
__rsub__() (object method)
__rtruediv__() (object method)
__rxor__() (object method)
__self__ (method attribute)
__set__() (object method)
__set_name__() (object method)
__setattr__() (object method)
__setitem__() (object method)
  (email.message.EmailMessage method)
  (email.message.Message method)
  (in module operator)
(method attribute)
(module attribute), [1]
(types.ModuleType attribute)
__enter__()  (contextmanager method)
  (object method)
  (winreg PyHKEY method)
__eq__()  (email.charset.Charset method)
  (email.header.Header method)
  (in module operator)
  (instance method)
  (memoryview method)
  (object method)
__excepthook__ (in module sys)
__exit__()  (contextmanager method)
  (object method)
  (winreg PyHKEY method)
__file__
  (module attribute), [1], [2]
__float__()  (object method)
__floor__()  (fractions.Fraction method)
  (object method)
__floordiv__() (in module operator)
  (object method)
__format__
__format__()  (datetime.date method)
  (datetime.datetime method)
  (datetime.time method)
  (email.charset.Charset method)
  (email.header.Header method)
  (email.headerregistry.Address method)
  (email.headerregistry.Group method)
  (email.message.EmailMessage method)
  (email.message.Message method)
  (multiprocessing.managers.BaseProxy method)
  (object method)
__sub__() (in module operator)
  (object method)
__subclasscheck__() (c
__subclasses__() (class
__subclasshook__() (method)
__suppress_context__ (traceback.TracebackException attribute)
__traceback__ (exception attribute)
__truediv__() (in module operator)
(mailbox.Mailbox method)
(mailbox.Maildir method)
(object method)
__setstate__() (copy protocol)
__slots__
__spec__
__stderr__ (in module sys)
__stdin__ (in module sys)
__stdout__ (in module sys)
__str__()  (datetime.date method)
  (datetime.datetime method)
  (datetime.time method)
  (email.charset.Charset method)
  (email.header.Header method)
  (email.headerregistry.Address method)
  (email.headerregistry.Group method)
  (email.message.EmailMessage method)
  (email.message.Message method)
  (multiprocessing.managers.BaseProxy method)
  (object method)
future statement
__future__ (module)
__future__ () (in module operator)
 (instance method)
 (object method)
__get__ () (object method)
__get_attribute__ (module attribute)
__getattribute__ () (object method)
__getitem__ () (object method)
(email.headerregistry.HeaderRegistry method)
 (email.message.EmailMessage method)
 (email.message.Message method)
 (in module operator)
 (mailbox.Mailbox method)
 (mapping object method)
 (object method)
 (re.Match method)
__getnewargs__ () (object method)
__getnewargs_ex__ () (object method)
__getstate__ () (copy protocol)
 (object method)
__globals__ (function attribute)
__gt__ () (in module operator)
 (instance method)
 (object method)
__hash__ () (object method)
__iadd__ () (in module operator)
 (object method)
__iand__ () (in module operator)
__xor__ () (in module operator)
 (object method)
__anonymous__ (ctypes.$
__asdict__ () (collection method)
_b_base__ (ctypes._CDa
_b_needsfree__ (ctypes.
_callmethod__ () (multiprocessing.manag
_CData (class in ctypes
_clear_type_cache__ () (in m
_current_frames__ () (in m
_debugmallocstats__ () (in
_dummy_thread (module
_enablelegacywindowsst
_exit__ () (in module os
_fields__ (collections.somena
_attribute)
_fields__ (ctypes.Structu
_fields_defaults (collections.somenamed
_flush__ () (wsgiref.ha
_frozen__ (C type)
_FuncPtr (class in ctyp
_get_child_mock__ () (m
_getframe__ () (in module:
_getvalue__ ()
(object method)
__iconcat__() (in module operator)
__ifloordiv__() (in module operator)
 (object method)
__ilshift__() (in module operator)
 (object method)
__imatmul__() (in module operator)
 (object method)
__imod__() (in module operator)
 (object method)
__import__
 built-in function
__import__() (built-in function)
 (in module importlib)
__imul__() (in module operator)
 (object method)
__index__() (in module operator)
 (object method)
__init__() (difflib.HtmlDiff method)
 (logging.Handler method)
 (logging.logging.Formatter method)
 (object method)
__init_subclass__() (object class method)
__instancecheck__() (class method)
__int__() (object method)
__interactivehook__ (in module sys)
__inv__() (in module operator)
__invert__() (in module operator)
 (object method)
__ior__() (in module operator)
 (object method)
__ipow__() (in module operator)
 (multiprocessing.managers.BaseProxy method)
__handle (ctypes.PyDLL method)
__initittab (C type)
__length__ (ctypes.Array attribute)
__locale module
__make__() (collection class method)
__makeResult() (unittest method)
__name (ctypes.PyDLL attribute)
__objects (ctypes._CData attribute)
__pack__ (ctypes.Structure attribute)
__parse() (gettext.NullTranslations method)
__Pointer (class in ctypes)
__Py_c_diff (C function)
__Py_c_neg (C function)
__Py_c_pow (C function)
__Py_c_prod (C function)
__Py_c_quot (C function)
__Py_c_sum (C function)
__Py__NoneStruct (C variable)
__PyBytes_Resize (C function)
__PyImport_Fini (C function)
__PyImport_Init (C function)
__PyObject_GC_TRACK (C function)
__PyObject_GC_UNTRACK (C function)
__PyObject_New (C function)
__PyObject_NewVar (C function)
__PyTuple_Resize (C function)
__replace() (collections.somenamedtuple method)
__setroot()
__irshift__() (in module operator)  
  (object method)  
__isub__() (in module operator)  
  (object method)  
__iter__() (container method)  
  (iterator method)  
  (mailbox.Mailbox method)  
  (object method)  
  (unittest.TestSuite method)  
__itruediv__() (in module operator)  
  (object method)  
__ixor__() (in module operator)  
  (object method)  

(xml.etree.ElementTree method)  
_SimpleCData (class in ctypes)  
_structure() (in module email.iterators)  
_thread  
  module  
_thread (module)  
_type_ (ctypes._Pointer)  
  (ctypes.Array)  
_write() (wsgiref.handlers.BaseHandler method)  
_xoptions (in module sys)
A (in module re)
A-LAW, [1]
a-LAW
a2b_base64() (in module binascii)
a2b_hex() (in module binascii)
a2b_hqx() (in module binascii)
a2b_qp() (in module binascii)
a2b_uu() (in module binascii)
a85decode() (in module base64)
a85encode() (in module base64)
ABC (class in abc)
abc (module)
ABCMeta (class in abc)
abiflags (in module sys)
abort() (asyncio.DatagramTransport method)
atabort() (asyncio.WriteTransport method)
(ftpplib.FTP method)
(in module os)
(threading.Barrier method)
above() (curses.panel.Panel method)
ABOVE_NORMAL_PRIORITY_CLASS (in module subprocess)
abs
built-in function, [1]
abs() (built-in function)
(decimal.Context method)
(in module operator)
abspath() (in module os.path)
abstract base class
AbstractAsyncContextManager (class in contextlib)
AbstractBasicAuthHandler (class in urllib.request)
(asyncio.Task method)
(concurrent.futures.Future method)
add_fallback() (gettext.NullTranslations method)
add_file() (msilib.Directory method)
add_flag() (mailbox.MaildirMessage method)
(add_mailbox.MMDFMessage method)
(add_mailbox.mboxMessage method)
add_flowing_data() (formatter.formatter method)
add_folder() (mailbox.Maildir method)
(mailbox.MH method)
add_get_handler()
(email.contentmanager.ContentManager method)
add_handler() (urllib.request.OpenerDirector
method)
add_header() (email.message.EmailMessage
method)
(email.message.Message method)
(urllib.request.Request method)
(wsgiref.headers.Headers method)
add_history() (in module readline)
add_hor_rule() (formatter.formatter method)
add_include_dir() (distutils.ccompiler.CCompiler
deb)
add_label() (mailbox.BabylMessage method)
add_label_data() (formatter.formatter method)
add_library() (distutils.ccompiler.CCompiler method)
add_library_dir() (distutils.ccompiler.CCompiler
method)
add_line_break() (formatter.formatter method)
add_link_object() (distutils.ccompiler.CCompiler
method)
add_literal_data() (formatter.formatter method)
(add_argparse.ArgumentParser method)
assert_not_called() (unittest.mock.Mock
method)
assert_python_failure() (in module test.support.script_helper)
assert_python_ok() (in module test.support.script_helper)
assertAlmostEqual() (unittest.TestCase
method)
assertCountEqual() (unittest.TestCase
method)
assertDictEqual() (unittest.TestCase
method)
assertEqual() (unittest.TestCase
method)
assertFalse() (unittest.TestCase
method)
assertGreater() (unittest.TestCase
method)
assertGreaterEqual() (unittest.TestCase
method)
assertIn() (unittest.TestCase
method)
assertIs() (unittest.TestCase
method)
assertIsInstance() (unittest.TestCase
method)
assertIsNone() (unittest.TestCase
method)
assertIsNot() (unittest.TestCase
method)
assertIsNotNone() (unittest.TestCase
method)
assertLess() (unittest.TestCase
method)
assertLessEqual() (unittest.TestCase
method)
assertListEqual() (unittest.TestCase
method)
assertLogs() (unittest.TestCase
method)
assertMultiLineEqual() (unittest.TestCase
method)
assertNotAlmostEqual() (unittest.TestCase
method)
addHandler() (logging.Logger method)
addition
addLevelName() (in module logging)
addnstr() (curses.window method)
AddPackagePath() (in module modulefinder)
addr (smtpd.SMTPChannel attribute)
addr_spec (email.headerregistry.Address attribute)
Address (class in email.headerregistry)
address (email.headerregistry.SingleAddressHeader attribute)
   (multiprocessing.connection.Listener attribute)
   (multiprocessing.managers.BaseManager attribute)
address_exclude() (ipaddress.IPv4Network method)
   (ipaddress.IPv6Network method)
address_family (socketserver.BaseServer attribute)
address_string()
addresses (email.headerregistry.AddressHeader attribute)
   (email.headerregistry.Group attribute)
AddressHeader (class in email.headerregistry)
addressof() (in module ctypes)
AddressValueError
addshape() (in module turtle)
addsitedir() (in module site)
addSkip() (unittest.TestResult method)
addstr() (curses.window method)
addSubTest() (unittest.TestResult method)
addSuccess() (unittest.TestResult method)
addTest() (unittest.TestSuite method)
addTests() (unittest.TestSuite method)
addTypeEqualityFunc() (unittest.TestCase method)
addUnexpectedSuccess()
method)
adjusted() (decimal.Decimal method)
adler32() (in module zlib)
ADPCM, Intel/DVI
adpcm2lin() (in module audioop)
AF_ALG (in module socket)
AF_CAN (in module socket)
AF_INET (in module socket)
AF_INET6 (in module socket)
AF_LINK (in module socket)
AF_PACKET (in module socket)
AF_RDS (in module socket)
AF_UNIX (in module socket)
AF_VSOCK (in module socket)
aifc (module)
aifc() (aifc.aifc method)
AIFF, [1]
aiff() (aifc.aifc method)
AIFF-C, [1]
alarm() (in module signal)
alaw2lin() (in module audioop)
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (in module ssl)
ALERT_DESCRIPTION_INTERNAL_ERROR (in module ssl)
AlertDescription (class in ssl)
algorithm_available (in module hashlib)
algorithms_guaranteed (in module hashlib)
alias (pdb command)
alignment() (in module ctypes)
alive (weakref.finalize attribute)
all() (built-in function)
all_errors (in module ftplib)
all_features (in module xml.sax.handler)
all_frames (tracemalloc.Filter attribute)
all_properties (in module xml.sax.handler)
all_suffixes () (in module importlib.machinery)
all_tasks() (asyncio.Task class method)
allocate_lock() (in module _thread)
allow_reuse_address (socketserver.BaseServer attribute)
allowed_domains() (http.cookiejar.DefaultCookiePolicy method)
alt() (in module curses.ascii)
ALT_DIGITS (in module locale)
altsep (in module os)
altzone (in module time)
ALWAYS_TYPED_ACTIONS (optparse.Option attribute)
AMPER (in module token)
AMPEREQUAL (in module token)
and
    bitwise
    operator, [1], [2]
and_() (in module operator)
annotated
    assignment
annotation
    (inspect.Parameter attribute)
annotations
    function, [1]
announce() (distutils.ccompiler.CCompiler method)
anonymous
    function
answer_challenge() (in module multiprocessing.connection)
anticipate_failure() (in module test.support)
Any (in module typing)
ANY (in module unittest.mock)
any() (built-in function)
AnyStr (in module typing)
api_version (in module sys)
apop() (poplib.POP3 method)
APPDATA
append() (array.array method)
   (collections.deque method)
   (email.header.Header method)
   (imaplib.IMAP4 method)
   (msilib.CAB method)
   (pipes.Template method)
   (sequence method)
   (xml.etree.ElementTree.Element method)
apend_history_file() (in module readline)

    AUDIO_FILE_ENCODING_LINEAR_16 (in module sunau)
    AUDIO_FILE_ENCODING_LINEAR_24 (in module sunau)
    AUDIO_FILE_ENCODING_LINEAR_32 (in module sunau)
    AUDIO_FILE_ENCODING_LINEAR_8 (in module sunau)
    AUDIO_FILE_ENCODING_MULAW_8 (in module sunau)
    AUDIO_FILE_MAGIC (in module sunau)
    AUDIODEV
    audioop (module)
augmented assignment
auth() (ftplib.FTP_TLS method)
   (smtplib.SMTP method)
authenticate() (imaplib.IMAP4 method)
AuthenticationError
authenticators() (netrc.netrc method)
autorename
authkey (multiprocessing.Process attribute)
auto (class in enum)
autorange()
avg() (in module audioop)
avgpp() (in module audioop)
avoids_symlink_attacks (shutil.rmtree attribute)
await
in comprehensions
awaitable
Awaitable (class in collections.abc)
(keyword
    class in typing)
b"  
   bytes literal
b'  
   bytes literal
b16decode() (in module base64)
b16encode() (in module base64)
b2a_base64() (in module binascii)
b2a_hex() (in module binascii)
b2a_hqx() (in module binascii)
b2a_qp() (in module binascii)
b2a_uu() (in module binascii)
b32decode() (in module base64)
b32encode() (in module base64)
b64decode() (in module base64)
b64encode() (in module base64)
b85decode() (in module base64)
b85encode() (in module base64)
Babyl (class in mailbox)
BabylMessage (class in mailbox)
back() (in module turtle)
backslash character
backslashreplace_errors() (in module codecs)
backup() (sqlite3.Connection method)
backward() (in module turtle)
BadStatusLine
BadZipFile
BadZipfile
Balloon (class in tkinter.tix)
Barrier (class in multiprocessing)
   (class in threading)
Barrier()
BOM_UTF32_LE (in module codecs)
BOM_UTF8 (in module codecs)
bool (built-in class)
Boolean
   object, [1]
   operation
   operations, [1]
   type
   values
BOOLEAN_STATE:  
   (configparser.ConfigParser attribute)
bootstrap() (in module ensurepip)
border() (curses.window method)
bottom() (curses.panel.Panel method)
bottom_panel() (in module curses.panel)
BoundArguments (class)
BoundaryError
BoundedSemaphore  
   (class in multiprocessing)
   (class in threading)
BoundedSemaphore()
   (multiprocessing.manager method)
box() (curses.window method)
bpformat() (bdb.Breakpoint method)
bpprint() (bdb.Breakpoint method)
break
   statement, [1],
break (pdb command)
break_anywhere() (pdb command)
break_here() (pdb command)
break_long_words
BasicInterpolation (class in configparser)
BasicTestRunner (class in test.support)
 baudrate() (in module curses)
bbox() (tkinter.ttk.Treeview method)
BDADDR_ANY (in module socket)
BDADDR_LOCAL (in module socket)
bdb
   module
Bdb (class in bdb)
bdb (module)
BdbQuit
BDFL
bdist_msi (class in distutils.command.bdist_msi)
beep() (in module curses)
Beep() (in module winsound)
BEFORE_ASYNC_WITH (opcode)
begin_fill() (in module turtle)
begin_poly() (in module turtle)
below() (curses.panel.Panel method)
BELOW_NORMAL_PRIORITY_CLASS (in module subprocess)
Benchmarking
benchmarking, [1], [2], [3]
betavariate() (in module random)
bgcolor() (in module turtle)
bgpic() (in module turtle)
bias() (in module audioop)
bidirectional() (in module unicodedata)
bigaddspacetest() (in module test.support)
BigEndianStructure (class in ctypes)
bigmemp test() (in module test.support)
bin() (built-in function)
binary
  arithmetic operation
  bitwise operation
  data, packing
  literals
Binary (class in msilib)
  (class in xmlrpc.client)

**binary file**
binary literal
binary mode
binary semaphores
BINARY_ADD (opcode)
BINARY_AND (opcode)
BINARY_FLOOR_DIVIDE (opcode)
BINARY_LSHIFT (opcode)
BINARY_MATRIX_MULTIPLY (opcode)
BINARY_MODULO (opcode)
BINARY_MULTIPLY (opcode)
BINARY_OR (opcode)
BINARY_POWER (opcode)
BINARY_RSHIFT (opcode)
BINARY_SUBSCR (opcode)
BINARY_SUBTRACT (opcode)
BINARY_TRUE_DIVIDE (opcode)
BINARY_XOR (opcode)
BinaryIO (class in typing)
binascii (module)
bind (widgets)
bind() (asyncore.dispatcher method)
  (inspect.Signature method)
  (socket.socket method)
bind_partial() (inspect.Signature method)
bind_port() (in module test.support)
bind_textdomain_codeset() (in module

**BUILD_TUPLE_UNPACK** (opcode)
**BUILD_TUPLE_UNPACK** (opcode)

**built-in**
  method
  types
**built-in function**
  __import__
  abs, [1]
  ascii
  bytes, [1]
  call
  chr
  classmethod
  compile, [1], [2]
  complex, [1]
  divmod, [1], [2]
  exec, [1], [2], ['
  float, [1], [2]
  hash, [1], [2], ['
  help
  id
  int, [1], [2]
  len, [1], [2], [3]
  [9], [10], [11]
  max
  min
  object, [1]
  open, [1]
  ord
  pow, [1], [2], [3]
  print
  range
gettext)
bind_unix_socket() (in module test.support)
binding
  global name
  name, [1], [2], [3], [4], [5], [6]
bindtextdomain() (in module gettext)
  (in module locale)
binhex
  module
binhex (module)
binhex() (in module binhex)
bisect (module)
bisect() (in module bisect)
bisect_left() (in module bisect)
bisect_right() (in module bisect)
bit_length() (int method)
bitmap() (msilib.Dialog method)
bitwise
  and
  operation, binary
  operation, unary
  operations
  or
  xor
bk() (in module turtle)
bkgd() (curses.window method)
bkgdset() (curses.window method)
blake2b() (in module hashlib)
blake2b, blake2s
blake2b.MAX_DIGEST_SIZE (in module hashlib)
blake2b.MAX_KEY_SIZE (in module hashlib)
repr, [1], [2], [3]
round
slice, [1]
staticmethod
tuple, [1]
type, [1], [2], [3]
built-in method
call
object, [1]
builtin_module_names (in module sys)
BuiltinFunctionType
BuiltinImporter
importlib.machinery
BuiltinMethodType
builtins
  module, [1], [2]
builtins (module)
ButtonBox (class in tkinter.tix)
bye() (in module turtle)
byref() (in module ctypes)
byte-code
  file, [1]
byte_compile() (in module dis)
bytearray
  formatting
  interpolation
  methods
  object, [1], [2],
  bytecode (built-in class)
  Bytecode (class in dis)
  Bytecode.codeobj (in module dis)
  Bytecode.first_line (in module dis)
blake2b.PERSON_SIZE  (in module hashlib)
blake2b.SALT_SIZE  (in module hashlib)
blake2s()  (in module hashlib)
blake2s.MAX_DIGEST_SIZE  (in module hashlib)
blake2s.MAX_KEY_SIZE  (in module hashlib)
blake2s.PERSON_SIZE  (in module hashlib)
blake2s.SALT_SIZE  (in module hashlib)
blank line
block
code
block_size  (hmac.HMAC attribute)
blocked_domains()
(http.cookiejar.DefaultCookiePolicy method)
BlockingIOError, [1]
blocksize  (http.client.HTTPConnection attribute)
BNF, [1]
body()  (nntplib.NNTP method)
body_encode()  (email.charset.Charset method)
body_encoding  (email.charset.Charset attribute)
body_line_iterator()  (in module email.iterators)
BOM  (in module codecs)
BOM_BE  (in module codecs)
BOM_LE  (in module codecs)
BOM_UTF16  (in module codecs)
BOM_UTF16_BE  (in module codecs)
BYTECODE_SUFFIXES  (in module importlib.machinery)
byteorder  (in module bytes)
built-in function
formatting
interpolation
methods
object, [1], [2]
str  (built-in class)
bytes  (built-in class)
(uuid.UUID attribute)
bytes literal
bytes-like object
bytes_le  (uuid.UUID attribute)
BytesFeedParser  (class)
BytesGenerator  (class)
email.generator
BytesHeaderParser  (class)
email.parser
BytesIO  (class in io)
BytesParser  (class)
ByteString  (class in typing)
byteswap()  (array.array method)
BytesWarning
bz2  (module)
BZ2Compressor  (class)
BZ2Decompressor
BZ2File  (class in bz
BOM_UTF16_LE (in module codecs)
BOM_UTF32 (in module codecs)
BOM_UTF32_BE (in module codecs)
C

C language, [1], [2], [3], [4], [5]
structures
C-contiguous, [1]
c_bool (class in ctypes)
C_BUILTIN (in module imp)
c_byte (class in ctypes)
c_char (class in ctypes)
c_char_p (class in ctypes)
c_contiguous (memoryview attribute)
c_double (class in ctypes)
C_EXTENSION (in module imp)
c_float (class in ctypes)
c_int (class in ctypes)
c_int16 (class in ctypes)
c_int32 (class in ctypes)
c_int64 (class in ctypes)
c_int8 (class in ctypes)
c_long (class in ctypes)
c_longdouble (class in ctypes)
c_longlong (class in ctypes)
c_short (class in ctypes)
c_size_t (class in ctypes)
c_ssize_t (class in ctypes)
c_ubyte (class in ctypes)
c_uint (class in ctypes)
c_uint16 (class in ctypes)
c_uint32 (class in ctypes)
c_uint64 (class in ctypes)
c_uint8 (class in ctypes)
c_ulong (class in ctypes)
c_ulonglong (class in ctypes)
colno (json.JSON
(re.error attri
cOLON (in modu
color() (in module
color_content() (in
color_pair() (in m
colormode() (in m
colorsys (module
COLS, [1]
column() (tkinter.t
COLUMNS, [1]
columns (os.term
combinations() (in
combinations_with
combiners)
combine() (dateti
combing() (in m
ComboBox (class
Combobox (class
comma
trailng
tuple display
COMMA (in modu
Command (class
(command (class in dist
command
(http.server.Base
attribute)
command line
command line op
--check-hash
default|alway
CALL_FUNCTION (opcode)
CALL_FUNCTION_EX (opcode)
CALL_FUNCTION_KW (opcode)
call_later() (asyncio.loop method)
call_list() (unittest.mock.call method)
CALL_METHOD (opcode)
call_soon() (asyncio.loop method)
call_soon_threadsafe() (asyncio.loop method)
call_tracing() (in module sys)
callable
   object, [1]
Callable (class in collections.abc)
   (in module typing)
callable() (built-in function)
CallableProxyType (in module weakref)
callback (optparse.Option attribute)
callback() (contextlib.ExitStack method)
callback_args (optparse.Option attribute)
callback_kwargs (optparse.Option attribute)
callbacks (in module gc)
called (unittest.mock.Mock attribute)
CalledProcessError
calloc()
CAN_BCM (in module socket)
can_change_color() (in module curses)
can_fetch()
   (urllib.robotparser.RobotFileParser method)
CAN_ISOTP (in module socket)
CAN_RAW_FD_FRAMES (in module socket)
can_symmlink() (in module test.support)
can_write_eof() (asyncio.StreamWriter method)
commenters (shlex.shlex attribute)
CommentHandler()
   (xml.parsers.expat.xmlparser method)
commit() (msilib.CAB method)
Commit() (msilib.Database method)
commit() (sqlite3.Connection method)
common (filecmp)
   common_dirs (filecmp)
   common_files (filecmp)
   common_funny (filecmp)
   commonpath() (in os)
   commonprefix() (in os)
communicate()
compare() (decimal.Context method)
   (subprocess)
   (decimal.Decimal method)
   (difflib.Differ method)
compare_digest() (in module hmac)
   (in module secrets)
compare_networks()
   (ipaddress.IFAddress)
   (ipaddress.IPv4Address)
   (ipaddress.IPv6Address)
COMPARE_OP (opcode)
cmp() (decimal.Decimal method)
cmp_to() (traceback.StackTraceSummary)
compare_signal()
   (decimal.Decimal method)
compare_to()
compare_total()
   (decimal.Decimal method)
compare_total_mag()
(asyncio.WriteTransport method)
can_xattr() (in module test.support)
cancel() (asyncio.Future method)
    (asyncio.Handle method)
    (asyncio.Task method)
    (concurrent.futures.Future method)
    (sched.scheduler method)
    (threading.Timer method)
cancel_dump_traceback_later() (in module faulthandler)
cancel_join_thread()
    (multiprocessing.Queue method)
cancelled() (asyncio.Future method)
    (asyncio.Handle method)
    (asyncio.Task method)
    (concurrent.futures.Future method)
CancelledError, [1]
CannotSendHeader
CannotSendRequest
canon() (bdb.Bdb method)
canonical() (decimal.Context method)
    (decimal.Decimal method)
capa() (poplib.POP3 method)
capitalize() (bytearray method)
    (bytes method)
    (str method)
Capsule
    object
captured_stderr() (in module test.support)
captured_stdin() (in module test.support)
captured_stdout() (in module test.support)
captureWarnings() (in module logging)
capwords() (in module string)
casefold() (str method)

comparing objects
comparison operator
COMPARISON_FLAGS
comparisons chaining, [1]
Compat32 (class in module python)
compat32 (in module email)
compile build-in function
Compile (class in module compile)
compile() (built-in)
    (distutils.ccompiler.CCompiler method)
    (in module py_compile)
    (in module re)
    (parser.ST method)
compile_command() (in module code)
compile_dir() (in module compileall)
compile_file() (in module compileall)
compile_path() (in module compileall)
compileall (module)
compileall command
    --invalidation
    hash|unchck

-b
-d destdir
-f
-i list
-j N
-l
-q
-r
cast() (in module ctypes)
  (in module typing)
    (memoryview method)
cat() (in module nis)
catch_warnings (class in warnings)
category() (in module unicodedata)
cbreak() (in module curses)
CC
ccc() (ftplib.FTP_TLS method)
CCompiler (class in distutils.ccompiler)
CDLL (class in ctypes)
ceil() (in module math), [1]
center() (bytearray method)
  (bytes method)
  (str method)
CERT_NONE (in module ssl)
CERT_OPTIONAL (in module ssl)
CERT_REQUIRED (in module ssl)
cert_store_stats() (ssl.SSLContext method)
cert_time_to_seconds() (in module ssl)
CertificateError
certificates
CFLAGS, [1], [2]
CFUNCTYPE() (in module ctypes)
CGI
debugging
  exceptions
  protocol
  security
  tracebacks
cgi (module)
cgi_directories
  (http.server.CGIHTTPRequestHandler attribute)
CGIHandler (class in wsgiref.handlers)  (ipaddress.IF
CGIHTTPRequestHandler (class in http.server) compression() (ss
cgi (module) CompressionError compressobj() (in
CGIXMLRPCRequestHandler (class in xmlrpc.server) COMSPEC, [1]
chain() (in module itertools) concat() (in module concatenation
chaining exception
collections) Condition (class in
typing) concurrent.futures:
ChainMap (class in collections) Condition (class in
(type in typing) multiprocessing):
change_cwd() (in module test.support) condition (pdb co
change_root() (in module distutils.util) condition) (msilib:
CHANNEL_BINDING_TYPES (in module ssl) Condition()
channel_class (smtpd.SMTPServer method) (multiprocessing.:
attribute) Conditional
collections) configParser (cla
channels() (ossaudiodev.oss_audio_device method) configuration
CHAR_MAX (in module locale) configParser (module
character, [1], [2] configuration file
CharacterDataHandler() file, debugger
(xml.parsers.expat.xmlparser method) file, path
Charset (class in email.charset) configuration info
Charset() (gettext.NullTranslations method) configure() (tkinte
characters() configure_mock()
(xml.sax.handler.ContentHandler method) confstr() (in module confstr_names (ir
characters_written (BlockingIOError attribute) conjugate() (com}
Charset (class in email.charset) configuration info
Charset() (gettext.NullTranslations method) configure() (tkinte
chdir() (in module os) configure_mock()
check (lzma.LZMADecompressor attribute) confstr() (in module confstr_names (ir
check() (imaplib.IMAP4 method) conjugate() (com
check_all() (in module test.support)
check_call() (in module subprocess)
check_environ() (in module distutils.util)
check_free_after_iterating() (in module test.support)
check_hostname (ssl.SSLContext attribute)
check_impl_detail() (in module test.support)
check_no_resource_warning() (in module test.support)
check_output() (doctest.OutputChecker method)
    (in module subprocess)
check_returncode() (subprocess.CompletedProcess method)
check_syntax_error() (in module test.support)
check_unused_args() (string.Formatter method)
check_warnings() (in module test.support)
checkbox() (msilib.Dialog method)
checkcache() (in module linecache)
CHECKED_HASH (py_compile.PycInvalidationMode attribute)
checkfuncname() (in module bdb)
CheckList (class in tkinter.tix)
checksizeof() (in module test.support)
checksum
    Cyclic Redundancy Check
chflags() (in module os)
chgat() (curses.window method)
childNodes (xml.dom.Node attribute)
ChildProcessError
test.support)
children (pyclbr.Class attribute)
chmod() (in module os)
(choice() (in module random)
(choices (optparse.Option attribute)
(choices() (in module random)
(chown() (in module os)
(chr
(chr() (built-in function)
(chroot() (in module os)
(Chunk (class in chunk)
(chunk (module)
cipher
(cipher() (ssl.SSLSocket method)
circle() (in module turtle)
CIRCUMFLEX (in module token)
CIRCUMFLEXEQUAL (in module token)
Clamped (class in decimal)
class
(attribute
(attribute assignment
(body
(constructor
(definition, [1]
(instance
(name
(object, [1], [2]
(statement
Class (class in symtable)
Class browser

consumed
attribute)
container, [1]
iteration over
Container (class i
(class in typi
contains() (in mod
content type
MIME
content_manager
attribute)
content_type
(email.headerregist
attribute)
ContentDispositiv
email.contentmanag
contents (ctypes.
contents() (i
method)
(ContentTooShortl
ContentTransferE
email.headerregist
ContentTypeHead
email.headerregist
Context (class in
(class in deci
context (ssl.SSLSo
context manager,
context_diff() (in r
class instance
    attribute
    attribute assignment
    call
    object, [1], [2]
class object
    call, [1], [2]
**class variable**
classmethod
    built-in function
classmethod() (built-in function)
ClassMethodDescriptorType (in module types)
ClassVar (in module typing)
clause
CLD_CONTINUED (in module os)
CLD_DUMPED (in module os)
CLD_EXITED (in module os)
CLD_TRAPPED (in module os)
clean() (mailbox.Maildir method)
cleandoc() (in module inspect)
CleanImport (class in test.support)
cleanup functions
clear (pdb command)
Clear Breakpoint
clear() (asyncio.Event method)
    (collections.deque method)
    (curses.window method)
    (dict method)
    (email.message.EmailMessage method)
    (frame method)
    (frozenset method)
    (http.cookiejar.CookieJar method)
    ContextDecorator
    contextlib (module)
    ContextManager
    contextmanager()
    ContextVar (class
    contextvars (mod
    contextvars.Token contigous, [1]
        (memoryview
    continue
        statement, [1
    continue (pdb cor
    CONTINUE_LOC
    Control (class in r
    (class in tkint
    control() (msilib.C
    (select.kqueue
    controlnames (in
    controls() (o method)
    conversion
        arithmetic
        string, [1
    ConversionError
    conversions
        numeric
    convert_arg_line_ (argparse.ArgumentParser
    convert_field() (st
    convert_path() (i
    Cookie (class in t
    CookieError
    CookieJar (class
    cookiejar (urllib.re
(in module turtle), [1]
(mailbox.Mailbox method)
(sequence method)
(threading.Event method)
(xml.etree.ElementTree.Element
method)
clear_all_breaks() (bdb.Bdb method)
clear_all_file_breaks() (bdb.Bdb method)
clear_bpbynumber() (bdb.Bdb method)
clear_break() (bdb.Bdb method)
clear_cache() (in module filecmp)
clear_content() (email.message.EmailMessage method)
clear_flags() (decimal.Context method)
clear_frames() (in module traceback)
clear_history() (in module readline)
clear_session_cookies() (http.cookiejar.CookieJar method)
clear_traces() (in module trace nuitro)'
clear_traps() (decimal.Context method)
clearcache() (in module linecache)
ClearData() (msilib.Record method)
clearok() (curses.window method)
clearscreen() (in module turtle)
clearstamp() (in module turtle)
clearstamps() (in module turtle)
Client() (in module multiprocessing.connection)
clock() (in module time)
CLOCK_BOOTTIME (in module time)
clock_getres() (in module time)
clock_gettime() (in module time)
clock_gettime_ns() (in module time)
CLOCK_HIGHRES (in module time)
CLOCK_MONOTONIC (in module time)
CLOCK_MONOTONIC_RAW (in module time)
CLOCK_PROCESS_CPUTIME_ID (in module time)
CLOCK_PROF (in module time)
CLOCK_REALTIME (in module time)
clock_settime() (in module time)
clock_settime_ns() (in module time)
CLOCK_THREAD_CPUTIME_ID (in module time)
CLOCK_UPTIME (in module time)
clone() (email.generator.BytesGenerator method)
    (email.generator.Generator method)
    (email.policy.Policy method)
    (in module turtle)
    (pipes.Template method)
cloneNode() (xml.dom.Node method)
close() (aifc.aifc method), [1]
    (asyncio.AbstractChildWatcher method)
    (asyncio.BaseTransport method)
    (asyncio.Server method)
    (asyncio.StreamWriter method)
    (asyncio.SubprocessTransport method)
    (asyncio.loop method)
    (asyncore.dispatcher method)
    (chunk.Chunk method)
    (contextlib.ExitStack method)
copy_sign() (decimal.Decimal method)
copy_tree() (in module distutils.dir_util)
copyfile() (in module shutil)
copyfileobj() (in module shutil)
copying files
copymode() (in module shutil)
copyreg (module)
copyright (built-in)
    (in module sys)
copysign() (in module math)
copystat() (in module shutil)
copytree() (in module shutil)
coroutine, [1], [2]
coroutine() (class in typing)
coroutine function
Coroutine (class in collections.abc)
coroutine function
coroutine()
    (in module asyncio)
    (in module types)
CoroutineType (in module types)
cos() (in module math)
cosh() (in module math)
count (tracemalloc.Statistic attribute)
count() (tracemalloc.StatisticDiff attribute)
count() (array.array method)
    (bytearray method)
    (bytes method)
    (collections.defaultdict method)
    (collections.deque method)
    (in module itertools)
    (sequence method)
    (str method)
method) (multiprocessing.Process method) (create_datagram_endpoint() in module asyncio)
(multiprocessing.Queue method) (create_decimal() in module decimal)
(multiprocessing.connection.Listener method) (create_decimal_from_float() in module decimal)
(multiprocessing.pool.Pool method) (create_default_context() in module ssl)
(os.scandir method) (CREATE_DEFAULT_ERROR_MODE in module subprocess)
(ossaludiodev.oss_audio_device method) (create_empty_file() in module test.support)
(ossaludiodev.oss_mixer_device method) (create_function() in module importlib)
(select.devpoll method) (create_future() in module asyncio)
(select.epoll method) (create_module() in module sqlite3)
(select.kqueue method) (create_module()) (importlib.machinery.ExtensionFileLoader)
(selectors.BaseSelector method) (create_module()) (importlib.abc.Loader)
(shelve.Shelf method) (importlib.machinery.ExtensionFileLoader)
(socket.socket method) (CREATE_NEW_CONSOLE in module subprocess)
(sqlite3.Connection method) (CREATE_NEW_CONSOLE in module subprocess)
(sqlite3.Cursor method) (create_shortcut() in distutils.core)
(sunau.AU_read method) (create_socket() in distutils.core)
(sunau.AU_write method) (create_subprocess_exec() in module asyncio)
(tarfile.TarFile method) (create_subprocess_shell() in module asyncio)
(telnetlib.Telnet method) (create_subprocess_shell() in module asyncio)
(urllib.request.BaseHandler method) (create_subprocess_shell() in module asyncio)
(wave.Wave_read method) (create_subprocess_shell() in module asyncio)
(wave.Wave_write method) (create_subprocess_shell() in module asyncio)
Close() (winreg.PyHKEY method) (create_subprocess_shell() in module asyncio)
close() (xml.etree.ElementTree.TreeBuilder method) (create_task() in module asyncio)
(xml.etree.ElementTree.XMLParser method)
(xml.etree.ElementTree.XMLPullParser method)
(xml.sax.xmlreader.IncrementalParser method)
(create_subprocess()) (distutils.core)
create_stats() (profile)
create_string_buffer() (ctypes)
create_subprocess() (distutils.core)
create_system() (zipfile)
create_task() (asyncio.loop)
**cmp_to_key() (in module functools)**
**cmpfiles() (in module filecmp)**
**CMSG_LEN() (in module socket)**
**CMSG_SPACE() (in module socket)**
**co_argcount (code object attribute)**
**CO_ASYNC_GENERATOR (in module inspect)**
**co_cellvars (code object attribute)**
**co_code (code object attribute)**
**co_consts (code object attribute)**
**CO_COROUTINE (in module inspect)**
**co_filename (code object attribute)**
**co_firstlineno (code object attribute)**
**co_flags (code object attribute)**
**co_freevars (code object attribute)**
**CO_FUTURE_DIVISION (C variable)**
**CO_GENERATOR (in module inspect)**
**CO_ITERABLE_COROUTINE (in module inspect)**
**co_lnotab (code object attribute)**
**co_name (code object attribute)**
**co_names (code object attribute)**
**CO_NESTED (in module inspect)**
**CO_NEWLOCALS (in module inspect)**
**co_nlocals (code object attribute)**
**CO_NOFREE (in module inspect)**
**CO_OPTIMIZED (in module inspect)**
**co_stacksize (code object attribute)**
**CO_VARARGS (in module inspect)**
**CO_VARKEYWORDS (in module inspect)**
**co_varnames (code object attribute)**
**code**
  **block**
**code (module)**
**crypt (module)**
**crypt() (in module crypt)**
**crypt(3), [1], [2]**
**cryptography**
**cssclass_month (attribute)**
**cssclass_month_head (attribute)**
**cssclass_noday (attribute)**
**cssclass_year (attribute)**
**cssclass_year_head (attribute)**
**cssclasses (calendar.HTMLCalendar.attribute)**
**cssclasses_week (calendar.HTMLCalendar.attribute)**
**csv**
  **(module)**
**cte**
  **(email.headerregistry.ContentTransferEncoding.attribute)**
**cte_type (email.policy.Policy.attribute)**
**ctermid() (in module os)**
**ctime() (datetime.datetime.method)**
**ctrl() (in module curses.ascii)**
**CTRL_BREAK_EVENT (in module signal)**
**CTRL_C_EVENT (in module signal)**
**ctypes (module)**
**curdir (in module os)**
**currency() (in module locale)**
(SystemExit attribute)
(urllib.error.HTTPError attribute)
(xml.etree.ElementTree.ParseError attribute)
(xml.parsers.expat.ExpatError attribute)
code object, [1], [2], [3]
code_info() (in module dis)
CodeInfo (class in codecs)
Codecs
decode
encode
codecs (module)
coded_value (http.cookies.Morsel attribute)
codeop (module)
codepoint2name (in module html.entities)
codes (in module xml.parsers.expat.errors)
CODESET (in module locale)
CodeType (in module types)
coding
    style
coeccion
col_offset (ast.AST attribute)
collapse_addresses() (in module ipaddress)
collapse/rfc2231_value() (in module email.utils)
collect() (in module gc)
collect_incoming_data()
(asynchat.async_chat method)
Collection (class in collections.abc)
    (class in typing)
collections (module)
collections.abc (module)
D

D_FMT (in module locale)
D_T_FMT (in module locale)
daemon (multiprocessing.Process attribute)
          (threading.Thread attribute)
dangling
d f l o f
data
    packing binary
    tabular
type
    type, immutable
Data (class in plistlib)
data (collections.UserDict attribute)
    (collections.UserList attribute)
    (collections.UserString attribute)
    (select.kevent attribute)
    (selectors.SelectorKey attribute)
    (urllib.request.Request attribute)
    (xml.dom.Comment attribute)
    (xml.dom.ProcessingInstruction attribute)
    (xml.dom.Text attribute)
    (xmlrpc.client.Binary attribute)
data()
(xml.etree.ElementTree.TreeBuilder method)
data_open()
    (urllib.request.DataHandler method)
data_received() (asyncio.Protocol method)
database

Dict (class in typing)
dict() (multiprocessing.managers.SyncManager method)
dictConfig() (in module logging.config)
dictionary
    display
    object, [1], [2], [3]
type, operations in dictionary view
    DictReader (class in csv)
    DictWriter (class in csv)
diff_bytes() (in module difflib)
diff_files (filecmp.dircmp attribute)
Differ (class in difflib),
difference() (frozenset method)
difference_update() (frozenset method)
difflib (module)
digest() (hashlib.hash)
    (hashlib.shake module)
    (hmac.HMAC method)
    (in module hmac)
digest_size (hmac.HMAC attribute)
digit() (in module unicodedata)
digits (in module string)
dir() (built-in function)
    (ftplib.FTP method)
dircmp (class in filecmp)
directory
    changing
    creating
    deleting, [1]
site-packages
traversal, [1] walking, [1]

Directory (class in ms directory)
(http.server.SimpleHTTP ...

DatumHandler (class in logging.handlers)
DatagramProtocol (class in asyncio)
DatagramRequestHandler (class in socketserver)
DatagramTransport (class in asyncio)
DataHandler (class in urllib.request)

date (class in datetime)
date() (datetime.datetime method)
   (ntplib.NNTP method)
date_time (zipfile.ZipInfo attribute)
date_time_string()
(http.server.BaseHTTPRequestHandler ...

DateHeader (class in email.headerregistry)
datetime (class in datetime)
DateTime (class in xmlrpc.client)
datetime
(email.headerregistry.DateHeader attribute)
   (module)
datum
day (datetime.date attribute)
   (datetime.datetime attribute)
day_abbr (in module calendar)
day_name (in module calendar)
daylight (in module time)
Daylight Saving Time
DbfilenameShelf (class in shelve)
dbm (module)
dbm.dumb (module)
dbm.gnu
   module, [1]
dbm.gnu (module)
dbm.ndbm
   module, [1]
dbm.ndbm (module)
dcgettext() (in module locale)
deallocation, object
debug (imaplib.IMAP4 attribute)
DEBUG (in module re)
debug (shlex.shlex attribute)
   (zipfile.ZipFile attribute)
debug() (in module doctest)
   (in module logging)
   (logging.Logger method)
   (pipes.Template method)
   (unittest.TestCase method)
   (unittest.TestSuite method)
DEBUG_BYTECODE_SUFFIXES (in module importlib.machinery)
DEBUG_COLLECTABLE (in module gc)
DEBUG_LEAK (in module gc)
debug_print()
   (distutils.ccompiler.CCompiler method)
DEBUG_SAVEALL (in module gc)
debug_src() (in module doctest)
DEBUG_STATS (in module gc)
DEBUG_UNCOLLECTABLE (in
   (mailbox.MH method)
   (mailbox.Mailbox method)
discard_buffers() (asyncore.dispatcher method)
disco() (in module dis)
discover() (unittest.TestLoader method)
disk_usage() (in module shutil)
dispatch_call() (bdb.Bdb method)
dispatch_exception() (bdb.Bdb method)
dispatch_line() (bdb.Bdb method)
dispatch_return() (bdb.Bdb method)
dispatch_table (pickle method)
dispatcher (class in asyncore)
dispatcher_with_send
   display
   dictionary
   list
   set
   tuple
   display (pdb command)
   display_name (email.headerregistry.Attribute method)
   (email.headerregistry.Group attribute)
displayhook() (in module sys)
dist() (in module platform)
distance() (in module turtle)
distb() (in module dis)
Distribution (class in distutils)
distutils (module)
distutils.archive_util (module)
distutils.bcppcompiler (module)
distutils.ccompiler (module)
distutils.cmd (module)
distutils.command (module)
distutils.cygpath (module)
distutils.command.bdist
module gc)

debugger, [1], [2]
  configuration file

debugging
  CGI
  assertions

DebuggingServer (class in smtpd)
debuglevel (http.client.HTTPResponse attribute)

DebugRunner (class in doctest)

Decimal (class in decimal)
decimal (module)
decimal literal
decimal() (in module unicodedata)

decode
  Codecs
  decode (codecs.CodecInfo attribute)
  decode() (bytearray method)
    (bytes method)
    (codecs.Codec method)
    (codecs.IncrementalDecoder method)
    (in module base64)
    (in module codecs)
    (in module quopri)
    (in module uu)
    (json.JSONDecoder method)
    (xmlrpc.client.Binary method)
    (xmlrpc.client.DateTime method)

decode_header() (in module email.header)
  (in module nntplib)

decode_params() (in module

distutils.command.bdist)
distutils.command.bdist_package

distutils.command.build

distutils.command.build_ext

distutils.core

distutils.cygwinnccompiler

distutils.debug

distutils.dep_util

distutils.dir_util

distutils.dist

distutils.errors

distutils.extension

distutils.fancy_getopt

distutils.file_util

distutils.filelist

distutils.log

distutils.msvccompiler

distutils.spawn
email.utils)
decode_rfc2231() (in module email.utils)
decode_source() (in module importlib.util)
decodebytes() (in module base64)
DecodedGenerator (class in email.generator)
decodestring() (in module base64)
    (in module quopri)
decomposition() (in module unicodedata)
decompress() (bz2.BZ2Decompressor method)
    (in module bz2)
    (in module gzip)
    (in module lzma)
    (in module zlib)
    (lzma.LZMADecompressor method)
    (zlib.Decompress method)
decompressobj() (in module zlib)
def
    statement
def_prog_mode() (in module curses)
def_shell_mode() (in module curses)
default
    parameter value
default (in module email.policy)
distutils.sysconfig (module)
distutils.text_file (module)
distutils.unixccompiler (module)
distutils.util (module)
distutils.version (module)
DISTUTILS_DEBUG divide() (decimal.Context method)
divide_int() (decimal.Context method)
DivisionByZero (class)
divmod
    built-in function, [1]
divmod() (built-in function)
    (decimal.Context method)
DllCanUnloadNow() (in module ctypes)
DllGetClassObject() (in module ctypes)
dllhandle (in module sys)
dngettext() (in module gettext)
do_clear() (bdb.Bdb method)
do_command() (curses.Textbox method)
do_GET() (http.server.SimpleHTTPRequestHandler method)
do_handshake() (sslrescia.method)
do_HEAD() (http.server.SimpleHTTPRequestHandler method)
do_POST() (http.server.CGIHTTPRequestHandler method)
doc (json.JSONDecodeError attribute)
doc_header (cmd.Cmd attribute)
DocCGIXMLRPCRequestHandler (class in xmlrpc.server)
DocFileSuite() (in module...
doCleanups() (unittest.TestCase method)
docmd() (smtplib.SMTP method)
docstring, [1]
doctest.DocTest docstrings, [1]
DocTest (class in doctest module)
DocTestFailure
DocTestFinder (class)
DocTestParser (class)
DocTestRunner (class)
DocTestSuite() (in module doctest)
doctype() (xml.etree.ElementTree method)
documentation generation
online documentation string
documentation strings
documentElement (xml.dom.Element attribute)
DocXMLRPCRequest (xmlrpc.server)
DocXMLRPCServer (class)
domain (email.headerregistry.Address attribute)
(tracemalloc.Dominion)
(tracemalloc.Filter attribute)
(tracemalloc.Trace attribute)
domain_initial_dot

domain_return_ok() (HTTPCookieJar method)
domain_specified

default () (configparser.ConfigParser method)
default_buffer_size (in module io)
default_factory (collections.defaultdict attribute)
default_exception_handler() (asyncio.loop method)
default_open() (urllib.request.BaseHandler method)
default_timer() (in module timeit)
DefaultContext (class in decimal)
DefaultCookiePolicy (class in http.cookiejar)
defaultdict (class in collections)
DefaultDict (class in typing)
DefaultEventLoopPolicy (class in asyncio)
DefaultHandler()
(xml.parsers.expat.xmlparser method)
DefaultHandlerExpand()
(xml.parsers.expat.xmlparser method)
defaults() (configparser.ConfigParser method)
default (inspect.Parameter attribute)
(optparse.Option attribute)
DEFAULT BUFFER_SIZE (in module io)
DEFAULT_FORMAT (in module tarfile)
DEFAULT_IGNORES (in module filecmp)
default_open() (urllib.request.BaseHandler method)
DEFAULT_PROTOCOL (in module pickle)
doctype() (xml.etree.ElementTree.TreeBuilder method)
documentElement (xml.dom.Document attribute)
DocXMLRPCRequestHandler (class in xmlrpc.server)
DocXMLRPCServer (class)
domain (email.headerregistry.Address attribute)
(tracemalloc.Dominion)
(tracemalloc.Filter attribute)
(tracemalloc.Trace attribute)
domain_initial_dot

domain_return_ok() (HTTPCookieJar method)
domain_specified

DEFAULT (in module unittest.mock)
DefaultSelector (class in selectors)  
defaultTestLoader (in module unittest)  
defaultTestResult() (unittest.TestCase method)  
defects  
(email.headerregistry.BaseHeader attribute)  
(email.message.EmailMessage attribute)  
(email.message.Message attribute)  
define_macro()  
(distutils.ccompiler.CCompiler method)  
definition  
    class, [1]  
    function, [1]  
defpath (in module os)  
DefragResult (class in urllib.parse)  
DefragResultBytes (class in urllib.parse)  
degrees() (in module math)  
    (in module turtle)  
del  
    statement, [1], [2], [3]  
delem  
(delattr() (built-in function)  
delay() (in module turtle)  
delay_output() (in module curses)  
delayload (http.cookiejar.FileCookieJar attribute)  
delch() (curses.window method)  
attribute)  
DomainFilter (class in tracemalloc)  
DomainLiberal (http.cookiejar.Default attribute)  
DomainRFC2965Match (http.cookiejar.Default attribute)  
DomainStrict (http.cookiejar.Default attribute)  
DomainStrictNoDots (http.cookiejar.Default attribute)  
DomainStrictNonDom (http.cookiejar.Default attribute)  
DOMEventStream (class in xml.dom.pulldom)  
DOMException  
DomstringSizeErr  
done() (asyncio.Future method)  
    (asyncio.Task method)  
    (concurrent.futures.Future method)  
    (in module turtle)  
    (xdrlib.Unpacker method)  
DONT_ACCEPT_BLANKLINE (in module doctest)  
DONT_ACCEPT_TRUE_FOR_1 (in module doctest)  
dont_write_bytecode (logging_handlers.RotationFileHandler method)  
doRollover()  
(logging_handlers.RotationFileHandler method)  
DOT (in module tokenize)  
dot() (in module turtle)  
DOTALL (in module re)  
doublequote (csv.Dialect attribute)  
DOUBLESLASH (in module token)
dele() (poplib.POP3 method)
delete() (ftplib.FTP method)
    (imaplib.IMAP4 method)
    (tkinter.ttk.Treeview method)
DELETE_ATTR (opcode)
DELETE_DEREF (opcode)
DELETE_FAST (opcode)
DELETE_GLOBAL (opcode)
DELETE_NAME (opcode)
DELETE_SUBSCR (opcode)
deleteacl() (imaplib.IMAP4 method)
deletefilehandler() (tkinter.Widget.tk method)
DeleteKey() (in module winreg)
DeleteKeyEx() (in module winreg)
deleln() (curses.window method)
deleteMe() (bdb.Breakpoint method)
DeleteValue() (in module winreg)
deletion
    attribute
    target
    target list
delimiter (csv.Dialect attribute)
delimiters
delitem() (in module operator)
deliver_challenge() (in module multiprocessing.connection)
delocalize() (in module locale)
demo_app() (in module wsgiref.simple_server)
denominator (fractions.Fraction attribute)
    (numbers.Rational attribute)
DeprecationWarning
DOUBLESLASHEQUAL (in module token)
DOUBLESTAR (in module token)
DOUBLESTAREQUAL (in module token)
doupdate() (in module curses)
down (pdb command)
down() (in module turtle)
drain() (asyncio.StreamWriter method)
drop_whitespace (attribute)
dropwhile() (in module itertools)
dst() (datetime.datetime method)
    (datetime.time method)
    (datetime.timezone method)
    (datetime.tzinfo method)
DTDHandler (class in xml.sax.handler)
DumbWriter (class in formatter)
dummy_threading (module)
dump() (in module ast)
    (in module json)
    (in module marshal)
    (in module pickle)
    (in module plistlib)
    (in module xml.etree.ElementTree)
    (pickle.Pickler method)
    (tracemalloc.Snapshot method)
dump_stats() (profile.Profile method)
    (pstats.Stats method)
dump_traceback() (in module faulthandler)
dump_traceback_later() (in module faulthandler)
dumps() (in module json)
    (in module marshal)
    (in module pickle)
    (in module plistlib)
deque (class in collections)
Deque (class in typing)
dequeue()
(logging.handlers.QueueListener method)
DER_cert_to_PEM_cert() (in module ssl)
derwin() (curses.window method)
DES
cipher
description (sqlite3.Cursor attribute)
description() (nntplib.NNTP method)
descriptions() (nntplib.NNTP method)
descritor
dest (optparse.Option attribute)
destructor, [1]
detach() (io.BufferedIOBase method)
   (io.TextIOBase method)
   (socket.socket method)
   (tkinter.ttk.Treeview method)
   (weakref.finalize method)
Detach() (winreg.PyHKEY method)
DETACHED_PROCESS (in module subprocess)
detect_api_mismatch() (in module test.support)
detect_encoding() (in module tokenize)
detect_language()
(distutils.ccompiler.CCompiler method)
deterministic profiling
device_encoding() (in module os)
devnull (in module os)
DEVNULL (in module subprocess)
devpoll() (in module select)
DevpollSelector (class in selectors)
dgettext() (in module gettext)
    (in module locale)
Dialect (class in csv)
dialect (csv.csvreader attribute)
    (csv.csvwriter attribute)
Dialog (class in msilib)
dict (2to3 fixer)
    (built-in class)
e
  in numeric literal
e (in module cmath)
    (in module math)
E2BIG (in module errno)
EACCES (in module errno)
EADDRINUSE (in module errno)
EADDRNOTAVAIL (in module errno)
EADV (in module errno)
EAFNOSUPPORT (in module errno)
E2BIG (in module errno)
EACCES (in module errno)
EADDRINUSE (in module errno)
EADDRNOTAVAIL (in module errno)
EADV (in module errno)
EAFNOSUPPORT (in module errno)
EAGAIN (in module errno)
EALREADY (in module errno)
EBADF (in module errno)
EAFNOSUPPORT (in module errno)
east_asian_width() (in module unicodedata)
EBADE (in module errno)
EBADF (in module errno)
EPROTOTYPE (in module errno)

environment
  deleting
  setting

EnvironmentError
  Environments
  virtual

EnvironmentVarGuard
  (class in test.support)

ENXIO (in module errno)
eof (bz2.BZ2Decompressor attribute)
(lzma.LZMADecompressor attribute)
(shlex.shlex attribute)
(ssl.MemoryBIO attribute)
(zlib.Decompress attribute)
eof_received ()
  (asyncio.BufferedProtocol method)
EOFError
  (built-in exception)
EOPNOTSUPP (in module errno)
EOVERFLOW (in module errno)
EPERM (in module errno)
EPFNOSUPPORT (in module errno)
epilogue

epoch

echo() (in module curses)
  (email)
echochar() (curses.window method)
ECHRN (in module errno)
ECOMM (in module errno)
ECONNABORTED (in module errno)
ECONNREFUSED (in module errno)
ECONNRESET (in module errno)
DEADLK (in module errno)
DEADLK (in module errno)
EDEADLOCK (in module errno)  eq() (in module operator)
EDESTADDRREQ (in module errno)  EQEQUAL
edit() (curses.textpad.Textbox method)  EQUAL (in module token)
EDOM (in module errno)  ERA (in module locale)
EDOTDOT (in module errno)  ERA_D_FMT
EDQUOT (in module errno)  ERA_D_T_FMT
EEXIST (in module errno)  ERANGE
EFAULT (in module errno)  EREMCHG
EDQUOT (in module errno)  EREMOTE
EHOSTDOWN (in module errno)  EREMOTE
EHOSTUNREACH (in module errno)  EREMOTE
EIDRM (in module errno)  errno
EILSEQ (in module errno)  errno (module OSError)
EINPROGRESS (in module errno)  errno (module urllib)
EINTR (in module errno)  error (arg error)
EINVAL (in module errno)  error (arg error)
EIO (in module errno)  error (arg error)
EISCONN (in module errno)  Error, [1], [11], [12]
EISDIR (in module errno)  error (arg error)
EISNAM (in module errno)  error (arg error)
EL2HLT (in module errno)  error (arg error)
EL2NSYNC (in module errno)  error (arg error)
EL3HLT (in module errno)  error (arg error)
EL3RST (in module errno)  error (arg error)
Element (class in xml.etree.ElementTree)  ElementDeclHandler()
  (xml.parsers.expat.xmlparser method) (loggit)
element_create() (tkinter.ttk.Style method)  elements() (collections.Counter method) (urllib)
ElementTree (class in xml.etree.ElementTree)
ELIBACC (in module errno)
ELIBBAD (in module errno)
ELIBEXEC (in module errno)
ELIBMAX (in module errno)
ELIBSCN (in module errno)
elif
    keyword
Ellinghouse, Lance
Ellipsis
    object
Ellipsis (built-in variable)
ELLIPSIS (in module doctest)
    (in module token)
ELNRNG (in module errno)
ELOOP (in module errno)
else
    conditional expression
    dangling
    keyword, [1], [2], [3], [4], [5]
email (module)
email.charset (module)
email.contentmanager (module)
email.encoders (module)
email.errors (module)
email.generator (module)
email.header (module)
email.headerregistry (module)
email.iterators (module)
email.message (module), [1]
email.mime (module)
email.parser (module)
email.policy (module)
email.utils (module)
evaluation order
Event (class (class event scheduling)
event() (method)
Event()
(multiprocessing method)
events (selector)
(widget)

enable_callback_tracebacks() (method)
enable_interspersed_args()
(optparse.OptionParser method)
enable_load_extension() (sqlite3.Connection method)
enable_traversal() (tkinter.ttk.Notebook method)
ENABLE_USER_SITE (module)
EnableReflectionKey() (module)
ENAMETOOLONG (module)
ENAVAIL (module)
enclose() (method)
encode
   Codecs
encode (codecs.CodecInfo attribute)
enable() (codecs.Codec method)
   (codecs.IncrementalEncoder method)
   (email.header.Header method)
   (base64 module)
   (codecs module)
   (quopri module)
   (uu module)
   (json.JSONEncoder method)
enable() (pdb command)
enable()
   (imaplib.IMAP4 method)
   (module)
   (faulthandler module)
   (module)
   (module)
   (module)
   (module)
enable_callback_tracebacks() (module)
enable_interspersed_args()
enable_load_extension() (module)
enable_traversal() (module)
ENABLE_USER_SITE (module)
EnableReflectionKey() (module)
ENAMETOOLONG (module)
ENAVAIL (module)
enclose() (method)
encode
   Codecs
encode (codecs.CodecInfo attribute)
enable() (codecs.Codec method)
   (codecs.IncrementalEncoder method)
   (email.header.Header method)
   (base64 module)
   (codecs module)
   (quopri module)
   (uu module)
   (json.JSONEncoder method)
enterabs() (sched.scheduler method)
entities (xml.dom.DocumentType attribute)
EntityDeclHandler() (xml.parsers.expat.xmlparser method)
extitydefs (in module html.entities)
EntityResolver (class in xml.sax.handler)
Enum (class in enum)
enum (module)
enum_certificates() (in module ssl)
enum_crls() (in module ssl)
enumerate() (built-in function)
    (in module threading)
EnumKey() (in module winreg)
EnumValue() (in module winreg)
EnvBuilder (class in venv)
environ (in module os)
    (in module posix)
environb (in module os)
environment
environment variable
    APPDATA
    AUDIODEV
    BROWSER, [1]
    CC
    CFLAGS, [1], [2]
    COLS, [1]
    COLUMNS, [1]
    COMSPEC, [1]
    CPP
    CPPFLAGS
    DISTUTILS_DEBUG
    HOME, [1], [2], [3], [4]
    HOMEDRIVE, [1]
    HOMEPATH, [1]
expand() (in module math)
expand_node() (xml.dom.pulldom.DOMEventStream method)
extnode() (xml.dom.pulldom.DOMEventStream method)
expm1() (in module math)
expovariate() (in module random)
expression
evaluatable
expvars() (in module os.path)
exptext()
fail() (unittest.expectedFailure)
failfast() (unittest.expectedFailure)
basestring (typing)
IDLESTARTUP
KDEDIR
LANG, [1], [2], [3], [4]
LANGUAGE, [1]
LC_ALL, [1]
LC_MESSAGES, [1]
LDCXXSHARED
LDFLAGS
LINES, [1], [2], [3], [4]
LNAME
LOGNAME, [1]
MIXERDEV
PAGER
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34]
PATHEXT, [1]
PLAT
POSIXLY_CORRECT
PYTHON*, [1], [2], [3]
PYTHONASYNCIODEBUG, [1], [2]
PYTHONBREAKPOINT, [1], [2], [3], [4]
PYTHONCASEOK, [1]
PYTHONCOERCECLOCALE, [1], [2]
PYTHONDEBUG, [1], [2]
PYTHONDEVMODE, [1], [2]
PYTHONDOCS
PYTHONDONTWRITEBYTECODE, [1], [2], [3], [4], [5], [6]
PYTHONDUMPREFS, [1]
PYTHONEXECUTABLE
PYTHONFAULTHANDLER, [1], [2]
PYTHONHASHSEED, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34]
SSL_CERT_PATH
SystemRoot
TCL_LIBRARY
TEMP
TERM, [1]
TK_LIBRARY
TMP
TMPDIR
TZ, [1], [2], [3], [4], [5]
USER
USERNAME, [1]
USERPROFILE, [1]
USER_BASE
exec_prefix, [1], [2]
http_proxy, [1], [2]
no_proxy
prefix, [1], [2], [3]
f'
    formatted string literal
f''
    formatted string literal
f-string, [1]
f_back (frame attribute)
f_builtins (frame attribute)
f_code (frame attribute)
f_contiguous (memoryview attribute)
f_globals (frame attribute)
f_lasti (frame attribute)
f_lineno (frame attribute)
f_locals (frame attribute)
F_LOCK (in module os)
F_OK (in module os)
F_TEST (in module os)
F_TLOCK (in module os)
f_trace (frame attribute)
f_trace_lines (frame attribute)
f_trace_opcodes (frame attribute)
F_ULOCK (in module os)
fabs() (in module math)
factorial() (in module math)
factory() (importlib.util.LazyLoader class method)
fail() (unittest.TestCase method)
FAIL_FAST (in module doctest)
failfast (unittest.TestResult attribute)
failureException (unittest.TestCase attribute)
failures (unittest.TestResult attribute)
FakePath (class in test.support)
False, [1], [2]

firstkey() (dbm.gnu.gdbm method)
firstweekday() (calendar method)
fix_missing_locations() (ast method)
fix_sentence_endings (textwrap.TextWrapper attribute)
Flag (class in enum)
flag_bits (zipfile.ZipInfo attribute)
flags (in module select)
flash() (in module curses)
flattening objects
float (built-in function)
float_info (in module sys)
float_repr_style (in module sys)
floating point literals
number objects
FloatingPoint
FloatOperation
floor() (in module math)
false
False (Built-in object)  floor() (in module math)
(built-in variable)  floordiv() (in module operator)
family (socket.socket attribute)  flush() (bz2.BZ2Compressor method)
fancy_getopt() (in module distutils.fancy_getopt)  (formatter.formatter method)
distutils.fancy_getopt)  (io.BufferedWriter method)
FancyGetopt (class in distutils.fancy_getopt)  (io.IOBase method)
FancyURLopener (class in urllib.request)  (logging.Logger method)
fast (pickle.Pickler attribute)  (logging.Logger method)
FastChildWatcher (class in asyncio)  (logging.Logger method)
fatalError() (xml.sax.handler.ErrorHandler method)
xml.sax.handler)
fauthandler (module)  (mailbox.Message method)
faultString (xmlrpc.client.Fault attribute)  (mailbox.Message method)
fchdir() (in module os)  (mmap.mmap method)
fchmod() (in module os)  (zlib.Compress method)
fchown() (in module os)  (zlib.Decompress method)
FCICreate() (in module msilib)  flush_headers()
fcntl (module)  (http.server.BaseHTTPRequestHandler method)
fcntl() (in module fcntl)  flush_softspace()
fd (selectors.SelectorKey attribute)  flushinp() (in mailbox)
fd() (in module turtle)  flushinp() (in mailbox)
fd_count() (in module test.support)  FlushKey() (in winreg)
fdataasync() (in module os)  fma() (decimal.Decimal method)
fdopen() (in module os)  fmod() (in module math)
Feature (class in msilib)  FMT_BINARY
feature_external_ges (in module xml.sax.handler)  FMT_XML
feature_external_pes (in module xml.sax.handler)  fnmatch (module)
feature_namespace_prefixes (in module xml.sax.handler)  fnmatchcase
mime.types modes object, [1] path configuration plist temporary

file (pyclbr.Class attribute) (pyclbr.Function attribute)

file ... compileall command line option

file control

unix

file name temporary

file object

io module open() built-in function

file-like object

FILE_ATTRIBUTE_ARCHIVE (in module stat) (traceback)
FILE_ATTRIBUTE_COMPRESSED (in module stat)
FILE_ATTRIBUTE_DEVICE (in module stat)
FILE_ATTRIBUTE_DIRECTORY (in module stat)
FILE_ATTRIBUTE_ENCRYPTED (in module stat)
FILE_ATTRIBUTE_HIDDEN (in module stat)
FILE_ATTRIBUTE_INTEGRITY_STREAM (in module stat)
FILE_ATTRIBUTE_NO_SCRUB_DATA (in module stat)
FILE_ATTRIBUTE_NORMAL (in module stat)
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (in module stat)

format() (in module locale) (logging.Formatter method)
format_datetime() (email.utils)
format_exc() (logging.Formatter method)
format_exception() (logging.Formatter method)
format_exception_only() (logging.Formatter method)
format_field() (string.Formatter method)
format_help() (argparse.ArgumentParser method)
format_list() (traceback)
format_map() (str method)
format_stack() (traceback)
format_stack_entry() (traceback)
format_string() (logging.Formatter method)
format_tb() (traceback)
format_usage() (argparse.ArgumentParser method)
FORMAT_VALUE (opcode)
formataddr() (email.utils)
FILE_ATTRIBUTE_OFFLINE (in module stat)
FILE_ATTRIBUTE_READONLY (in module stat)
FILE_ATTRIBUTE_REPARSE_POINT (in module stat)
FILE_ATTRIBUTE_SPARSE_FILE (in module stat)
FILE_ATTRIBUTE_SYSTEM (in module stat)
FILE_ATTRIBUTE_TEMPORARY (in module stat)
FILE_ATTRIBUTE_VIRTUAL (in module stat)
file_created() (built-in function)
file_dispatcher (class in asyncore)
file_open() (urllib.request.FileHandler method)
file_size (zipfile.ZipInfo attribute)
file_wrapper (class in asyncore)
filecmp (module)
fileConfig() (in module logging.config)
FileCookieJar (class in http.cookiejar)
FileEntry (class in tkinter.tix)
FileExistsError
FileFinder (class in importlib.machinery)
FileHandler (class in logging)
    (class in urllib.request)
FileInput (class in fileinput)
fileinput (module)
FileIO (class in io)
filelineno() (in module fileinput)
FileLoader (class in importlib.abc)
filemode() (in module stat)
filename (doctest.DocTest attribute)
    ( OSError attribute)
    (http.cookiejar.FileCookieJar attribute)
    (traceback.TracebackException attribute)
formatargspec
formatargvalues
formatdate()
FormatError
FormatError()
formatException()
formatStack()
formatmonth()
(calendar.HTMLCalendar method)
formatyear()
(calendar.HTMLCalendar method)
formatStack()
formatted string
Formatter (class in string)
formatTime()
formatWarning()
formatyearpage()
(calendar.HTMLCalendar method)
formatting
bytarray
bytes (%)
formatting, string
formatwarning()
formatyear()
formatyearrange
forward() (in module turtle)
found_terminator()
(foreground.fontwidth attribute)
fqdn (smtpd.SMTPChannel attribute)
Fortran contiguous
found_terminator() (asynchat.async_chat method)
found_terminator() (asyncore.dispatcher method)
(tracemalloc.Frame attribute)
(zipfile.ZipFile attribute)
(zipfile.ZipInfo attribute)
filename() (in module fileinput)
filename2 (OSError attribute)
filename_only (in module tabnanny)
filename_pattern (tracemalloc.Filter attribute)
filenames
  pathname expansion
  wildcard expansion
fileno() (http.client.HTTPResponse method)
  (in module fileinput)
  (io.IOBase method)
  (multiprocessing.connection.Connection method)
  (ossaudiodev.oss_audio_device method)
  (ossaudiodev.oss_mixer_device method)
  (select.devpoll method)
  (select.epoll method)
  (select.kqueue method)
  (selectors.DevpollSelector method)
  (selectors.EpollSelector method)
  (selectors.KqueueSelector method)
  (socket.socket method)
  (socketserver.BaseServer method)
  (telnetlib.Telnet method)
FileNotFoundError
fileobj (selectors.SelectorKey attribute)
FileSelectBox (class in tkinter.tix)
FileType (class in argparse)
FileWrapper (class in wsgiref.util)
fill() (in module textwrap)
  (textwrap.TextWrapper method)
fillcolor() (in module turtle)
filling() (in module turtle)
filter (2to3 fixer)
Filter (class in logging)
    (class in tracemalloc)
filter (select.kevent attribute)
filter() (built-in function)
    (in module curses)
    (in module fnmatch)
    (logging.Filter method)
    (logging.Handler method)
    (logging.Logger method)
FILTER_DIR (in module unittest.mock)
filter_traces() (tracemalloc.Snapshot method)
filterfalse() (in module itertools)
filterwarnings() (in module warnings)
finalization, of objects
finalize (class in weakref)
finalize_options() (distutils.cmd.Command method)
finalizer
finally
    keyword, [1], [2], [3], [4]
find() (bytearray method)
    (bytes method)
    (doctest.DocTestFinder method)
    (in module gettext)
    (mmap.mmap method)
    (str method)
    (xml.etree.ElementTree.Element method)
    (xml.etree.ElementTree.ElementTree method)
find_class() (pickle protocol)
    (pickle.Unpickler method)
find_library() (in module ctypes.util)
find_library_file() (distutils.ccompiler.CCompiler method)

find_loader() (importlib.abc.PathEntryFinder method)
  (importlib.machinery.FileFinder method)
  (in module importlib)
  (in module pkgutil)

find_longest_match() (difflib.SequenceMatcher method)

find_module() (imp.NullImporter method)
  (importlib.abc.Finder method)
  (importlib.abc.MetaPathFinder method)
  (importlib.abc.PathEntryFinder method)
  (importlib.machinery.PathFinder class method)
  (in module imp)
  (zipimport.zipimporter method)

find_msvcrt() (in module ctypes.util)

find_spec (importlib.abc.MetaPathFinder method)
  (importlib.abc.PathEntryFinder method)
  (importlib.machinery.FileFinder method)
  (importlib.machinery.PathFinder class method)
  (in module importlib.util)

find_unused_port() (in module test.support)

find_user_password()
  (urllib.request.HTTPPasswordMgr method), [1]

findall() (in module re)
  (re.Pattern method)
  (xml.etree.ElementTree.Element method)
  (xml.etree.ElementTree.ElementTree method)

find_loader() (importlib.abc.PathEntryFinder method)
  (importlib.machinery.FileFinder method)
  (in module importlib)
  (in module pkgutil)

find_longest_match() (difflib.SequenceMatcher method)

find_module() (imp.NullImporter method)
  (importlib.abc.Finder method)
  (importlib.abc.MetaPathFinder method)
  (importlib.abc.PathEntryFinder method)
  (importlib.machinery.PathFinder class method)
  (in module imp)
  (zipimport.zipimporter method)

find_msvcrt() (in module ctypes.util)

find_spec (importlib.abc.MetaPathFinder method)
  (importlib.abc.PathEntryFinder method)
  (importlib.machinery.FileFinder method)
  (importlib.machinery.PathFinder class method)
  (in module importlib.util)

find_unused_port() (in module test.support)

find_user_password()
  (urllib.request.HTTPPasswordMgr method), [1]

findall() (in module re)
  (re.Pattern method)
  (xml.etree.ElementTree.Element method)
  (xml.etree.ElementTree.ElementTree method)
method)
findCaller() (logging.Logger method)
finder, [1]
    find_spec
Finder (class in importlib.abc)
findfactor() (in module audioop)
findfile() (in module test.support)
findfit() (in module audioop)
finditer() (in module re)
    (re.Pattern method)
findlabels() (in module dis)
findlinestarts() (in module dis)
findmatch() (in module mailcap)
findmax() (in module audioop)
findtext() (xml.etree.ElementTree.Element method)
    (xml.etree.ElementTree.ElementTree method)
finish() (socketserver.BaseRequestHandler method)
finish_request() (socketserver.BaseServer method)
firstChild (xml.dom.Node attribute)

FTP_TLS (class)
FTPHandler (class)
ftplib (module)
fttruncate() (in module Full)
full() (asyncio.Queue method)
    (multiprocessing.Queue method)
    (queue.Queue method)
full_url (attribute)
fullmatch() (in module re)
    (re.Pattern method)
func (functools.partial attribute)
funcattrs (2to3 fixer)
function annotation
    anonymous argument
call, [1], [2] call, user-definition
generator
ame, [1] object, [1]
user-defined Function (class)
function and
function__entry (C function)
function__return (C function)
FunctionTestCase (class)
FunctionType
functools (module)
funny_files (filecmp.dircmp attribute)
future
statement

future (2to3 fixer)

Future (class in asyncio)
(class in concurrent.futures)

FutureWarning

fwalk() (in module os)
G

G.722
gaierror

gamma() (in module math)
gammavariate() (in module random)
garbage (in module gc)
garbage collection, [1]
gather() (curses.textpad.Textbox method)
gauss() (in module random)
gc (module)

gc__done (C function)
gc__start (C function)
gc_collect() (in module test.support)
gcd() (in module fractions)

    (in module math)
ge() (in module operator)
gen_lib_options() (in module distutils.ccompiler)
gen_preprocess_options() (in module distutils.ccompiler)
gen_uuid() (in module msilib)
generate_help()

    (distutils.fancy_getopt.FancyGetopt method)
gen, [1]

    expression
    function, [1], [2]
    iterator, [1]
    object, [1], [2]
Generator (class in collections.abc)

    (class in email.generator)
    (class in typing)
genenerator expression, [1]
generator iterator
generatorExit
exception, [1]

GeneratorType (in module types)

generic

    special attribute

Generic (class in typing)

generic function

generic_visit() (ast.NodeVisitor method)
genops() (in module pickletools)

get() (asyncio.Queue method)

    (configparser.ConfigParser method)

    (contextvars.Context method)

    (contextvars.ContextVar method)

    (dict method)

    (email.message.EmailMessage method)

    (email.message.Message method)

    (in module webbrowser)

    (mailbox.Mailbox method)

    (multiprocessing.Queue method)

    (multiprocessing.SimpleQueue method)

    (multiprocessing.pool.AsyncResult method)

    (ossaudiodev.oss_mixer_device method)

    (queue.Queue method)

    (queue.SimpleQueue method)

    (tkinter.ttk.Combobox method)

    (tkinter.ttk.Spinbox method)

    (types.MappingProxyType method)

    (xml.etree.ElementTree.Element method)

GET_AITER (opcode)

get_all() (email.message.EmailMessage method)

    (email.message.Message method)

    (wsgiref.headers.Headers method)

get_all_breaks() (bdb.Bdb method)

get_all_start_methods() (in module multiprocessing)

getcomptype()

getctime() (in module os.path)

getcwd() (in module os)

getcwdu (2to3 fixer)

getdecoder()

gettext()

gettext()

getctime() (in module os.path)

getcwd() (in module os)

getcwdb() (in module os)

getdefaultencoding()

getdefaultlocale()

getdefaulttimeout()

getdlopenflags()

getdoc()

getDOMImplementation() (in module xml.dom)

getDTDHandler()

getEffectiveLevel() (logging.Logger method)

getegid() (in module os)

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()

togethertags()
GET_ANEXT (opcode)
get_app()  (wsgiref.simple_server.WSGIServer method)
get_archive_formats() (in module shutil)
get_asyncgen_hooks() (in module sys)
get_attribute() (in module test.support)
GET_AWAITABLE (opcode)
get_begidx() (in module readline)
get_blocking() (in module os)
get_body()  (email.message.EmailMessage method)
get_body_encoding() (email.charset.Charset method)
get_boundary()  (email.message.EmailMessage method)
get_break() (bdb.Bdb method)
get_breaks() (bdb.Bdb method)
get_buffer() (asyncio.BufferedProtocol method)
get_bytes() (mailbox.Mailbox method)
get_ca_certs() (ssl.SSLContext method)
get_cache_token() (in module abc)
get_channel_binding() (ssl.SSLSocket method)
get_charset() (email.message.Message method)
get_charsets()  (email.message.EmailMessage method)
get_child_watcher() (asyncio.AbstractEventLoopPolicy method)
get_children() (symtable.SymbolTable method)
getEncoding() (xml.sax.xmlreader.InputSource method)
getEntityResolver() (xml.sax.xmlreader.XMLReader method)
getenv() (in module os)
getenvb() (in module os)
getErrorHandler() (xml.sax.xmlreader.XMLReader method)
geteuid() (in module os)
getEvent() (log.sys method)
getEventCategory() (logging.handlers.NTEventLogHandler method)
getException() (xml.sax.SAXException method)
getFeature() (xml.sax.xmlreader.XMLReader method)
GetFieldCount() (msilib.Record method)
getfile() (in module inspect)
getfirst() (configparser.ConfigParser method)
getfmts() (ossaudiodev.oss_audio_device method)
getfqdn() (in module socket)
getframeinfo() (in module inspect)
(tkinter.ttk.Treeview method)
get_ciphers() (ssl.SSLContext method)
get_clock_info() (in module time)
get_close_matches() (in module difflib)
get_code() (importlib.abc.InspectLoader method)
   (importlib.abc.SourceLoader method)
   (importlib.machinery.ExtensionFileLoader method)
   (importlib.machinery.SourcelessFileLoader method)
   (zipimport.zipimporter method)
get_completer() (in module readline)
get_completer_delims() (in module readline)
get_completion_type() (in module readline)
get_config_h_filename() (in module distutils.sysconfig)
   (in module sysconfig)
get_config_var() (in module distutils.sysconfig)
   (in module sysconfig)
get_config_vars() (in module distutils.sysconfig)
   (in module sysconfig)
get_content()
   (email.contentmanager.ContentManager method)
      (email.message.EmailMessage method)
         (email.message.Message method)
get_content_charset()
   (email.message.EmailMessage method)
   (email.message.Message method)
get_content_disposition()
   (email.message.EmailMessage method)
      (email.message.Message method)
get_content_maintype()
   (email.message.EmailMessage method)
      (email.message.Message method)
get_content_subtype()
(email.message.EmailMessage method)
(email.message.Message method)
get_content_type()
(email.message.EmailMessage method)
(email.message.Message method)
get_context() (in module multiprocessing)
get_coroutine_origin_tracking_depth() (in module sys)
get_coroutine_wrapper() (in module sys)
get_count() (in module gc)
get_current_history_length() (in module readline)
get_data() (importlib.abc.FileLoader method)
(importlib.abc.ResourceLoader method)
(in module pkgutil)
(zipimport.zipimporter method)
get_date() (mailbox.MaildirMessage method)
get_debug() (asyncio.loop method)
(in module gc)
get_default() (argparse.ArgumentParser method)
get_default_compiler() (in module distutils.ccompiler)
get_default_domain() (in module nis)
get_default_type()
(email.message.EmailMessage method)
(email.message.Message method)
get_default_verify_paths() (in module ssl)
get_dialect() (in module csv)
get_docstring() (in module ast)
get_doctest() (doctest.DocTestParser method)
get_endidx() (in module readline)
get_environ()
(wsgiref.simple_server.WSGIRequestHandler method)
get_environment()
(xml.etree.ElementTree method)
getitimer()
(in module signal)
getkey() (curses.window method)
GetLastError() (in module ctypes)
getLength() (xml.sax.xmlreader.Attributes method)
getLevelName() (in module logging)
getline() (in module linecache)
getLineNumber() (xml.sax.xmlreader.Locator method)
getlist() (cgi.FieldStorage method)
getloadavg() (in module os)
getlocale() (in module locale)
getLogger() (in module logging)
getLoggerClass() (in module logging)
getlogin() (in module os)
getLogRecordFactory() (in module logging)
getmark() (aifc.aifc method)
(sunau.AU_read method)
(wave.Wave_read method)
getmarkers() (aifc.aifc method)
(sunau.AU_read method)
(wave.Wave_read method)
getmaxyx() (in module curses)
getmember() (tarfile.TarFile method)
getmessage() (xml.etree.ElementTree method)
getMessage() (logging.LogRecord method)
(xml.sax.SAXException method)
getMessageID() (logging.handlers.NTEventLogHandler method)
get_errno() (in module ctypes)
get_event_loop()
  (asyncio.AbstractEventLoopPolicy method)
  (in module asyncio)
get_event_loop_policy() (in module asyncio)
get_examples() (doctest.DocTestParser method)
get_exception_handler() (asyncio.loop method)
get_exec_path() (in module os)
get_extra_info() (asyncio.BaseTransport method)
  (asyncio.StreamWriter method)
get_field() (string.Formatter method)
get_file() (mailbox.Babyl method)
  (mailbox.MH method)
  (mailbox.MMDF method)
  (mailbox.Mailbox method)
  (mailbox.Maildir method)
  (mailbox.mbox method)
get_file_breaks() (bdb.Bdb method)
get_filename() (email.message.EmailMessage method)
  (email.message.Message method)
  (importlib.abc.ExecutionLoader method)
  (importlib.abc.FileLoader method)
  (importlib.machinery.ExtensionFileLoader method)
  (zipimport.zipimporter method)
get_flags() (mailbox.MaildirMessage method)
  (mailbox.MMDFMessage method)
  (mailbox.mboxMessage method)
get_folder() (mailbox.Maildir method)
  (mailbox.MH method)
get_frees() (symtable.Function method)
get_freeze_count() (in module gc)
get_from() (mailbox.mboxMessage method)
get_map() (selectors.BaseSelector method)
get_matching_blocks() (difflib.SequenceMatcher method)
get_message() (mailbox.Mailbox method)
get_method() (urllib.request.Request method)
get_methods() (symtable.Class method)
get_mixed_type_key() (in module ipaddress)
get_name() (symtable.Symbol method)
    (symtable.SymbolTable method)
get_namespace() (symtable.Symbol method)
get_namespaces() (symtable.Symbol method)
get_nonstandard_attr() (http.cookiejar.Cookie method)
get_nowait() (asyncio.Queue method)
    (multiprocessing.Queue method)
    (queue.Queue method)
    (queue.SimpleQueue method)
get_object_traceback() (in module tracemalloc)
get_objects() (in module gc)
get_opcodes() (difflib.SequenceMatcher method)
get_option() (optparse.OptionParser method)
get_option_group() (optparse.OptionParser method)
get_option_order() (distutils.fancy_getopt.FancyGetopt method)
get_original_stdout() (in module test.support)
get_osfhandle() (in module msvcrt)
get_output_charset() (email.charset.Charset method)
get_param() (email.message.Message method)
get_parameters() (symtable.Function method)
get_params() (email.message.Message method)
get_path() (in module sysconfig)
method)
    (importlib.abc.SourceLoader method)
    (importlib.machinery.ExtensionFileLoader method)
    (importlib.machinery.SourcelessFileLoader method)
    (zipimport.zipimporter method)
get_special_folder_path() (built-in function)
get_stack() (asyncio.Task method)
    (bdb.Bdb method)
get_start_method() (in module multiprocessing)
get_starttag_text() (html.parser.HTMLParser method)
get_stats() (in module gc)
get_stderr() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
get_stdin() (wsgiref.handlers.BaseHandler method)
    (wsgiref.simple_server.WSGIRequestHandler method)
get_string() (mailbox.Mailbox method)
get_subdir() (mailbox.MaildirMessage method)
get_suffixes() (in module imp)
get_symbols() (symtable.SymbolTable method)
get_tag() (in module imp)
get_task_factory() (asyncio.loop method)
get_terminal_size() (in module os)
    (in module shutil)
get_terminator() (asynchat.async_chat method)
get_threshold() (in module gc)
get_token() (shlex.shlex method)
get_traceback_limit() (in module tracemalloc)
get_traced_memory() (in module tracemalloc)
get_tracemalloc_memory() (in module tracemalloc)
get_timeout() (socket.socket method)
gettrace() (in module sys)
getturtle() (in module turtle)
getType() (xml.sax.xmlreader.Attributes method)
get_value() (xml.sax.xmlreader.Attributes method)
get_value() (xml.sax.xmlreader.Attributes method)
get_tag() (in module imp)
get_task_factory() (asyncio.loop method)
get_terminal_size() (in module os)
    (in module shutil)
get_terminator() (asynchat.async_chat method)
get_threshold() (in module gc)
get_token() (shlex.shlex method)
get_traceback_limit() (in module tracemalloc)
get_traced_memory() (in module tracemalloc)
get_tracemalloc_memory() (in module tracemalloc)
grouping
  groups
  (email.headerregistry.AddressHeader attribute)
  (re.Pattern attribute)
  groups() (re.Matcher)
  grp (module grp)
  gt() (in module)
  guess_all_extensions()
  (mimetypes.MimeTypes method)
  guess_extension()
  (mimetypes.MimeTypes method)
  guess_scheme()
  (wsgiref.util.guess_scheme())
  guess_type()
  (mimetypes.MimeTypes method)
GUI
gzip (module gzip)
GzipFile (class in gzip)
halfdelay() (in module curses)
Handle (class in asyncio)
handle an exception
handle()  (http.server.BaseHTTPRequestHandler method)
  (logging.Handler method)
  (logging.Formatter method)
  (logging.NullHandler method)
  (logging.handlers.QueueListener method)
  (socketserver.BaseRequestHandler method)
  (wsgiref.simple_server.WSGIRequestHandler method)
handle_accept() (asyncore.dispatcher method)
handle_accepted() (asyncore.dispatcher method)
handle_charref()  (html.parser.HTMLParser method)
handle_close() (asyncore.dispatcher method)
handle_comment()  (html.parser.HTMLParser method)
handle_connect() (asyncore.dispatcher method)
handle_data() (html.parser.HTMLParser method)
handle_decl() (html.parser.HTMLParser method)
handle_defect() (email.policy.Policy method)
handle_endtag()  (html.parser.HTMLParser method)
handle_entityref() (html.parser.HTMLParser method)
handle_error() (asyncore.dispatcher method)
  (socketserver.BaseServer method)
handle_expect_100()  (http.server.BaseHTTPRequestHandler method)
handle_expt() (asyncore.dispatcher method)
handle_one_request()
(http.server.BaseHTTPRequestHandler method)
handle_pi() (html.parser.HTMLParser method)
handle_read() (asyncore.dispatcher method)
handle_request() (socketserver.BaseServer method)
(xmlrpc.server.CGIXMLRPCRequestHandler method)
handle_startendtag() (html.parser.HTMLParser method)
handle_starttag() (html.parser.HTMLParser method)
handle_timeout() (socketserver.BaseServer method)
handle_write() (asyncore.dispatcher method)
handleError() (logging.Handler method)
(logging.handlers.SocketHandler method)
handler exception Handler (class in logging)
handler() (in module cgitb)
harmonic_mean() (in module statistics)
HAS_ALPN (in module ssl)
has_children() (symtable.SymbolTable method)
has_colors() (in module curses)
HAS_ECDH (in module ssl)
has_exec() (symtable.SymbolTable method)
has_extn() (smtplib.SMTP method)
has_function() (distutils.ccompiler.CCompiler method)
has_header() (csv.Sniffer method)
(urllib.request.Request method)
has_ic() (in module curses)
has_il() (in module curses)
has_ipv6 (in module socket)
has_key (2to3 fixer)
has_key() (in module curses)
has_location (importlib.machinery.ModuleSpec attribute)
HAS_NEVER_CHECK_COMMON_NAME (in module ssl)
has_nonstandard_attr() (http.cookiejar.Cookie method)
HAS_NPN (in module ssl)
has_option() (configparser.ConfigParser method)
(hasoptparse.OptionParser method)
has_section() (configparser.ConfigParser method)
HAS_SNI (in module ssl)
HAS_SSLv2 (in module ssl)
HAS_SSLv3 (in module ssl)
has_ticket (ssl.SSLSession attribute)
HAS_TLSv1 (in module ssl)
HAS_TLSv1_1 (in module ssl)
HAS_TLSv1_2 (in module ssl)
HAS_TLSv1_3 (in module ssl)
hasattr() (built-in function)
hasAttribute() (xml.dom.Element method)
hasAttributeNS() (xml.dom.Element method)
hasAttributes() (xml.dom.Node method)
hasChildNodes() (xml.dom.Node method)
has_free (in module dis)
has_const (in module dis)
has_feature() (xml.dom.DOMImplementation method)
has_free (in module dis)
hash
    built-in function, [1], [2], [3]
hash character

hash
hash() (built-in function)

hash-based pyc

hash.block_size (in module hashlib)

hash.digest_size (in module hashlib)

hash_info (in module sys)

hashable, [1]

Hashable (class in collections.abc)

(has in typing)

hashHandlers() (logging.Logger method)

hashlib (module)

hashjabs (in module dis)

hashjrel (in module dis)

haslocal (in module dis)

hasname (in module dis)

HAVE_ARGUMENT (opcode)

HAVE_DOCSTRINGS (in module test.support)

HAVE_THREADS (in module decimal)

HCI_DATA_DIR (in module socket)

HCI_FILTER (in module socket)

HCI_TIME_STAMP (in module socket)

head() (nntplib.NNTP method)

Header (class in email.header)

header_encode() (email.charset.Charset method)

header_encode_lines() (email.charset.Charset method)

header_encoding (email.charset.Charset attribute)

header_factory (email.policy.EmailPolicy attribute)

header_fetch_parse() (email.policy.Compat32 method)

(header.policy.EmailPolicy method)

(header.policy.Policy method)

header_items() (urllib.request.Request method)

header_max_count() (email.policy.EmailPolicy HTTP_P HALF...
method)
    (email.policy.Policy method)
header_offset (zipfile.ZipInfo attribute)
header_source_parse()  (email.policy.Compat32 method)
    (email.policy.EmailPolicy method)
    (email.policy.Policy method)
header_store_parse()  (email.policy.Compat32 method)
    (email.policy.EmailPolicy method)
    (email.policy.Policy method)
HeaderError
HeaderParseError
HeaderParser (class in email.parser)
HeaderRegistry (class in email.headerregistry)
headers
    MIME, [1]
Headers (class in wsgiref.headers)
headers  (http.server.BaseHTTPRequestHandler attribute)
    (urllib.error.HTTPError attribute)
    (xmlrpc.client.ProtocolError attribute)
heading() (in module turtle)
    (tkinter.ttk.Treeview method)
heapify() (in module heapq)
heapmin() (in module msvcrt)
heappop() (in module heapq)
heappush() (in module heapq)
HTTPSHandler (class in urllib.request)
HTTPStatus (class in http)
hypot() (in math)
I (in module re)
I/O control
  POSIX
  UNIX
  buffering, [1]
  tty
iadd() (in module operator)
iand() (in module operator)
iconcat() (in module operator)
id
  built-in function
id (ssl.SSLSession attribute)
id() (built-in function)
  (unittest.TestCase method)
idcok() (curses.window method)
ident (select.kevent attribute)
  (threading.Thread attribute)
identchars (cmd.Cmd attribute)
identifier, [1]
identify()  (tkinter.ttk.Notebook method)
  (tkinter.ttk.Treeview method)
  (tkinter.ttk.Widget method)
identify_column()
  (tkinter.ttk.Treeview method)
identify_element()
  (tkinter.ttk.Treeview method)
identify_region()  (tkinter.ttk.Treeview method)
identify_row()  (tkinter.ttk.Treeview method)
identity
IOBase (class in io)
ioctl() (in module fcntl)
  (socket.socket method)
IOCTL_VM_SOCKETS(module socket)
IOError
ior() (in module operator)
  (ipaddress.IPv4Interface attribute)
  (ipaddress.IPv6Interface attribute)
  (ipaddress.ip_address() (in module
  ip_interface() (in module
  ip_network() (in module
  ipaddress (module)
  ipow() (in module
  ipv4_mapped (ipaddress:
  IPv4Address (class in ip
  IPv4Interface (class in ip:
  IPv4Network (class in ip
  IPV6_ENABLED (in module
  IPv6Address (class in ip
  IPv6Interface (class in ip:
  IPv6Network (class in ip
  irshift() (in module operator
is
  operator, [1]
is not
  operator, [1]
is_() (in module operator
is_absolute() (pathlib.PurePath
is_alive() (multiprocessing.
  (threading.Thread
is_android (in module te
test
identity of an object
test identities (2to3 fixer)
IDLE, [1]
IDLE_PRIORITY_CLASS (in module subprocess)
IDLESTARTUP
idlok() (curses.window method)
if
    conditional expression
    in comprehensions
    statement, [1]
if_indextoname() (in module socket)
if_nameindex() (in module socket)
if_nametoindex() (in module socket)
ifloordiv() (in module operator)
iglob() (in module glob)
ignoreWhitespace()
(xml.sax.handler.ContentHandler method)
ignore (pdb command)
ignore_errors() (in module codecs)
IGNORE_EXCEPTION_DETAIL (in module doctest)
ignore_patterns() (in module shutil)
IGNORECASE (in module re)
ihave() (ntplib.NNTP method)
IISCGIHandler (class in wsgiref.handlers)
ilshift() (in module operator)
imag (numbers.Complex attribute)
imaginary literal
imap() (multiprocessing.pool.Pool method)
is_assigned() (symtable.Symbol method)
is_attachment() (email.mime.message method)
is_authenticated() (urllib.request.HTTPPassword method)
is_block_device() (pathlib.Path method)
is_blocked() (http.cookies.Cookie method)
is_canonical() (decimal.Decimal method)
is_char_device() (pathlib.Path method)
IS_CHARACTER_JUNK (in module difflib)
is_check_supported() (in module lzma)
is_closed() (asyncio.stream.StreamWriter method)
is_closing() (asyncio.BaseTransport method)
is_dataclass() (in module dataclasses)
is_declared_global() (symtable.Symbol method)
is_dir() (os.DirEntry method)
is_enabled() (in module faulthandler)
is_expired() (http.cookies.Cookie method)
is_fifo() (pathlib.Path method)
is_file() (os.DirEntry method)
is_finalizing() (in module sys)
is_finite() (decimal.Decimal method)
is_free() (symtable.Symbol method)
is_global (ipaddress.IPv4Address attribute)
is_global (ipaddress.IPv6Address attribute)
is_global() (symtable.Symbol method)
IMAP4
  protocol
IMAP4 (class in imaplib)
IMAP4.abort
IMAP4.error
IMAP4.readonly
IMAP4_SSL
  protocol
IMAP4_SSL (class in imaplib)
IMAP4_stream
  protocol
IMAP4_stream (class in imaplib)
imap_unordered()
  (multiprocessing.pool.Pool method)
imaplib (module)
imatmul() (in module operator)
imghdr (module)
immedok() (curses.window method)
immutable
data type
  object, [1], [2]
  sequence types
immutable object
immutable sequence
  object
immutable types
  subclassing
imod() (in module operator)
imp
  module
imp (module)
ImpImporter (class in pkgutil)
impl_detail() (in module test.support)
implementation (in module sys)
is_hop_by_hop() (in module wsgiref.util)
is_imported() (symtable.Symbol method)
is_infinite() (decimal.Context method)
is_integer() (float method)
is_jython (in module test.support)
IS_LINE_JUNK() (in module difflib)
is_link_local (ipaddress.IPv4Address attribute)
is_loopback (ipaddress.IPv4Network attribute)
is_local() (symtable.Symbol method)
is_multicast (ipaddress.IPv4Address attribute)
is_multicast (ipaddress.IPv4Network attribute)
is_multicast (ipaddress.IPv6Address attribute)
is_multicast (ipaddress.IPv6Network attribute)
is_multipart() (email.message.EmailMessage method)
is_multipart() (email.message.Message method)
is_namespace() (symtable.Symbol method)
is_nan() (decimal.Context method)
is_normal() (decimal.Decimal method)
is_nested() (symtable.Symbol method)
is_normal() (decimal.Decimal method)
is_not() (in module operator)
is_not_allowed() (http.cookiejar.DefaultCookiePolicy method)
ImpLoader (class in pkgutil)

import
    hooks
    statement, [1], [2], [3], [4]
import (2to3 fixer)
import hooks
import machinery

**import path**
import__find__load__done (C function)
import__find__load__start (C function)
import_fresh_module() (in module test.support)
IMPORT_FROM (opcode)
import_module() (in module importlib)
    (in module test.support)
IMPORT_NAME (opcode)
IMPORT_STAR (opcode)

**importer**
ImportError
    exception

**importing**
importlib (module)
importlib.abc (module)
importlib.machinery (module)
importlib.resources (module)
importlib.util (module)
imports (2to3 fixer)
imports2 (2to3 fixer)
ImportWarning
ImproperConnectionState
imul() (in module operator)

is_optimized() (symtable.SymbolTable method)
is_package() (importlib.abc.SOURCE_MODULE method)
is_private (ipaddress.IPv4Address attribute)
is_private (ipaddress.IPv4Network attribute)
is_private (ipaddress.IPv6Address attribute)
is_private (ipaddress.IPv6Network attribute)
is_python_build() (in module sysconfig)
is_qnan() (decimal.Decimal method)
is_reading() (asyncio.ReadTransport method)
is_referenced() (symtable.Symbol method)
is_reserved (ipaddress.IPv4Address attribute)
is_reserved (ipaddress.IPv4Network attribute)
is_reserved (ipaddress.IPv6Address attribute)
is_reserved (ipaddress.IPv6Network attribute)
is_resource() (importlib.abc.ResourceReader method)
is_resource_enabled() (in module importlib)
is_running() (asyncio.loop method)
is_safe (uuid.UUID attribute)
is_serving() (asyncio.Server method)
is_set() (asyncio.Event method)
in keyword operator, [1], [2]
in_dll() (ctypes._CData method)
in_table_a1() (in module stringprep)
in_table_b1() (in module stringprep)
in_table_c11() (in module stringprep)
in_table_c11_c12() (in module stringprep)
in_table_c12() (in module stringprep)
in_table_c21() (in module stringprep)
in_table_c21_c22() (in module stringprep)
in_table_c22() (in module stringprep)
in_table_c3() (in module stringprep)
in_table_c4() (in module stringprep)
in_table_c5() (in module stringprep)
in_table_c6() (in module stringprep)
in_table_c7() (in module stringprep)
in_table_c8() (in module stringprep)
in_table_c9() (in module stringprep)
in_table_d1() (in module stringprep)
in_table_d2() (in module stringprep)
in_transaction (sqlite3.Connection attribute)
inch() (curses.window method)
inclusive
  or
inclusive (tracemalloc.DomainFilter attribute)
  (tracemalloc.Filter attribute)
Incomplete
IncompleteRead
IncompleteReadError
is_signed() (decimal.Context method)
  (decimal.Decimal method)
is_site_local (ipaddress.IPv6Network attribute)
is_snan() (decimal.Context method)
  (decimal.Decimal method)
is_socket() (pathlib.Path method)
is_subnormal() (decimal.Decimal method)
is_symmlink() (os.DirEntry method)
  (pathlib.Path method)
is_tarfile() (in module tarfile)
is_term_resized() (in module curses)
is_tracing() (in module tracemalloc)
is_tracked() (in module gc)
is_unspecified
  (ipaddress.IPv4Network attribute)
  (ipaddress.IPv6Address attribute)
  (ipaddress.IPv6Network attribute)
is_wintouched() (curses.window method)
is_zero() (decimal.Context method)
  (decimal.Decimal method)
is_zipfile() (in module zipfile)
isabs() (in module os.path)
isabstract() (in module os.path)
IsADirectoryError
isalnum() (bytearray method)
  (bytes method)
  (str method)
isalpha() (bytearray method)
  (bytes method)
incr_item(), [1]
increment_lineno() (in module ast)
IncrementalDecoder (class in codecs)
incrementaldecoder (codecs.CodecInfo attribute)
IncrementalEncoder (class in codecs)
incrementalencoder (codecs.CodecInfo attribute)
IncrementalNewlineDecoder (class in io)
IncrementalParser (class in xml.sax.xmlreader)
ident (doctest.Example attribute)
INDENT (in module token)
INDENT token
indent() (in module textwrap)
indentation
IndentationError
index operation
index() (array.array method)
  (bytearray method)
  (bytes method)
  (collections.deque method)
  (in module operator)
  (sequence method)
  (str method)
  (tkinter.ttk.Notebook method)
  (tkinter.ttk.Treeview method)
IndexError
indexOf() (in module operator)
IndexSizeErr
indices() (slice method)
inet_aton() (in module socket)
inet_ntoa() (in module socket)
inet_ntop() (in module socket)
inet_pton() (in module socket)
Inexact (class in decimal)
inf (in module cmath)
infile
  json.tool command line option
infile (shlex.shlex attribute)
Infinity
infj (in module cmath)
info() (dis.Bytecode method)
  (gettext.NullTranslations method)
  (in module logging)
  (logging.Logger method)
infolist() (zipfile.ZipFile method)
inheritance
ini file
init() (in module mimetypes)
init_color() (in module curses)
init_database() (in module msilib)
init_pair() (in module curses)
inited (in module mimetypes)
initgroups() (in module os)
initial_indent (textwrap.TextWrapper attribute)
initialize_options()
  (distutils.cmd.Command method)
initscr() (in module curses)
inode() (os.DirEntry method)
INPLACE_ADD (opcode)
INPLACE_AND (opcode)
isdir() (in module os.path)
isdisjoint() (frozenset method)
isdown() (in module turtle)
isEnabled() (in module x
isDebugEnabled() (logging.
isendwin() (in module curses)
ISEOF() (in module tokenize)
isexpr() (in module parser)
  (parser.ST method)
isfifo() (tarfile.TarInfo method)
isfile() (in module os.path)
  (tarfile.TarInfo method)
isfinite() (in module cmath)
  (in module math)
isfirstline() (in module file)
isframe() (in module inspect)
isfunction() (in module inspect)
isfuture() (in module asyncio)
isgenerator() (in module inspect)
isgeneratorfunction() (in module inspect)
isgetsetdescriptor() (in module inspect)
isgraph() (in module curses)
isidentifier() (str method)
isinf() (in module cmath)
  (in module math)
isinstance (2to3 fixer)
isinstance() (built-in function)
iskeyword() (in module keyword)
isleap() (in module calendar)
islice() (in module itertools)
islink() (in module os.path)
islnk() (tarfile.TarInfo method)
INPLACE_FLOOR_DIVIDE (opcode)
INPLACE_LSHIFT (opcode)
INPLACE_MATRIX_MULTIPLY (opcode)
INPLACE_MODULO (opcode)
INPLACE_MULTIPLY (opcode)
INPLACE_OR (opcode)
INPLACE_POWER (opcode)
INPLACE_RSHIFT (opcode)
INPLACE_SUBTRACT (opcode)
INPLACE_TRUE_DIVIDE (opcode)
INPLACE_XOR (opcode)

input
   (2to3 fixer)
input() (built-in function)
   (in module fileinput)
input_charset (email.charset.Charset attribute)
input_codec  (email.charset.Charset attribute)
InputOnly (class in tkinter.tix)
InputSource (class in xml.sax.xmlreader)
inquiry (C type)
insch() (curses.window method)
insdelln() (curses.window method)
insert() (array.array method)
   (collections.deque method)
   (sequence method)
   (tkinter.ttk.Notebook method)
   (tkinter.ttk.Treeview method)
   (xml.etree.ElementTree.Element method)
insert_text() (in module readline)
insertBefore() (xml.dom.Node method)
insertln() (curses.window method)
insnstr() (curses.window method)
insort() (in module bisect)
insort_left() (in module bisect)
insort_right() (in module bisect)
inspect (module)
inspect command line option
   --details
InspectLoader (class in importlib.abc)
insstr() (curses.window method)
install() (gettext.NullTranslations method)
   (in module gettext)
install_opener() (in module urllib.request)
install_scripts() (venv.EnvBuilder method)
installHandler() (in module unittest)
instance
call, [1]
class
object, [1], [2]
instancemethod
   object
instate() (tkinter.ttk.Widget method)
instr() (curses.window method)
instream (shlex.shlex attribute)
Instruction (class in dis)
Instruction.arg (in module dis)
Instruction.argrepr (in module dis)
Instruction.argval (in module dis)
Instruction.is_jump_target (in module dis)
Instruction.offset (in module dis)
Instruction.opcode (in module dis)
Instruction.opname (in module dis)
Instruction.starts_line (in module dis)

int
    built-in function, [1], [2]
int (built-in class)
    (uuid.UUID attribute)
Int2AP() (in module imaplib)
int_info (in module sys)

integer
    literals
    object, [1], [2]
    representation
    types, operations on

integer literal
Integral (class in numbers)
Integrated Development Environment
IntegrityError
Intel/DVI ADPCM
IntEnum (class in enum)
interact (pdb command)
interact() (code.InteractiveConsole method)
    (in module code)
    (telnetlib.Telnet method)

iterable
unpacking
Iterable (class in collections)
    (class in typing)

iterator
Iterator (class in collections)
    (class in typing)

iterdecode() (in module codecs)
iterdir() (pathlib.Path method)
iterdump() (sqlite3.Connection method)
iterencode() (in module codecs)

intern (2to3 fixer)
intern() (in module sys)
internal type
internal_attr (zipfile.ZipInfo attribute)
Internaldate2tuple() (in module imaplib)
internalSubset
(xml.dom.DocumentType attribute)
Internet
interpolated string literal
interpolation
  bytarray (%)
  bytes (%)
interpolation, string (%)
InterpolationDepthError
InterpolationError
InterpolationMissingOptionError
InterpolationSyntaxError
interpreted
interpreter
interpreter lock
interpreter prompts
interpreter shutdown
interpreterRequires_environment()
(in module test.support.script_helper)
interrupt() (sqlite3.Connection method)
interrupt_main() (in module _thread)
InterruptedError
intersection() (frozenset method)
intersection_update() (frozenset method)
IntFlag (class in enum)
(json.JSONEncoder method)
iterfind() (xml.etree.Element method)
iteritems() (mailbox.Mail
iterkeys() (mailbox.Mail
itermonthdates() (calendar
itermonthdays() (calendar
itermonthdays2() (calendar
itermonthdays3() (calendar
itermonthdays4() (calendar
interoparse() (in module xml.etree.ElementTree method)
itertext() (xml.etree.ElementTree method)
itertools (2to3 fixer)
(itertools module)
itertools_imports (2to3 fixer)
itervalues() (mailbox.Mail
iterweekdays() (calendar
ITIMER_PROF (in module
ITIMER_REAL (in module
ITIMER_VIRTUAL (in module
ItimerError
itruediv() (in module operator
ixor() (in module operator
intro (cmd.Cmd attribute)
InuseAttributeErr
inv() (in module operator)
InvalidAccessErr
invalidate_caches()
(importlib.abc.MetaPathFinder method)
  (importlib.abc.PathEntryFinder method)
  (importlib.machinery.FileFinder method)
  (importlib.machinery.PathFinder class method)
  (in module importlib)
InvalidCharacterErr
InvalidModificationErr
InvalidOperation (class in decimal)
InvalidStateErr
InvalidStateError
InvalidURL
inversion
invert() (in module operator)
invocation
io
  module
IO (class in typing)
io (module)
io.StringIO
  object
in numeric literal

Jansen, Jack

Java language

d=java_ver() (in module platform)
join() (asyncio.Queue method)
    (bytearray method)
    (bytes method)
    (in module os.path)
    (multiprocessing.JoinableQueue method)
    (multiprocessing.Process method)
    (multiprocessing.pool.Pool method)
    (queue.Queue method)
    (str method)
    (threading.Thread method)
    join_thread() (in module test.support)
        (multiprocessing.Queue method)
JoinableQueue (class in multiprocessing)

joinpath() (pathlib.PurePath method)
js_output()
    (http.cookies.BaseCookie method)
    (http.cookies.Morsel method)

json module
json (module)
json.tool (module)
json.tool command
    option
        --sort-keys
        -h, --help
        infile
        outfile

JSONDecodeError
JSONDecoder (class in json)
JSONEncoder (class in json)

jump (pdb command)
JUMP_ABSOLUTE (opcode)
JUMP_FORWARD (opcode)
JUMP_IF_FALSE_OR_POP (opcode)
JUMP_IF_TRUE_OR_POP (opcode)
kbhit() (in module msvcr100)
KDEDIR
kevent() (in module select)
key
    (http.cookies.Morsel attribute)
**key function**
key/datum pair
KEY_ALL_ACCESS (in module winreg)
KEY_CREATE_LINK (in module winreg)
KEY_CREATE_SUB_KEY (in module winreg)
KEY_ENUMERATE_SUB_KEYS (in module winreg)
KEY_EXECUTE (in module winreg)
KEY_NOTIFY (in module winreg)
KEY_QUERY_VALUE (in module winreg)
KEY_READ (in module winreg)
KEY_SET_VALUE (in module winreg)
KEY_WOW64_32KEY (in module winreg)
KEY_WOW64_64KEY (in module winreg)
KEY_WRITE (in module winreg)
KeyboardInterrupt (built-in exception), [1]
KeyError

**keyword**
as, [1], [2]
async
await, [1]
elif
else, [1], [2], [3], [4],
except
finally, [1], [2], [3], [4],
from, [1], [2], [3], [4],
in
yield

**keyword (module)**

**keyword argument**
keywords (functools.partial)
kil() (asyncio.asyncio.subprocess method)
    (asyncio.SubprocessTransport method)
    (in module os)
    (multiprocessing.Process method)
    (subprocess.Popen)
kill_python() (in test.support.script_helper)
killchar() (in module curses)
killpg() (in module os)
kind (inspect.Parameter)
knownfiles (in module mimetypes)
kqueue() (in module select)
keyrefs() (weakref.WeakKeyDictionary method)
keys() (contextvars.Context method)

KqueueSelector (class in selectors)
kwvars (inspect.BoundArguments attribute)
kwlist (in module keyword)
L (in module re)
LabelEntry (class in tkinter.tix)
LabelFrame (class in tkinter.tix)
**lambda**
  expression, [1]
  form
LambdaType (in module types)
LANG, [1], [2], [3], [4]
LANGUAGE, [1]
language
  C, [1], [2], [3], [4], [5]
  Java
large files
LargeZipFile
last() (ntplib.NNTP method)
last_accepted
  (multiprocessing.connection.Listener attribute)
last_traceback (in module sys), [1]
last_type (in module sys)
last_value (in module sys)
lastChild (xml.dom.Node attribute)
lastcmd (cmd.Cmd attribute)
lastgroup (re.Match attribute)
lastindex (re.Match attribute)
lastResort (in module logging)
lastrowid (sqlite3.Cursor attribute)
layout() (tkinter.ttk.Style method)
lazycache() (in module linecache)
LazyLoader (class in importlib.util)
LBRACE (in module token)
**LBYL**
LC_ALL, [1]
LC_COLLATE (in module locale) (importlib.abc.InspectLoader method)
LC_CTYPE (in module locale) (importlib.abc.Loader method)
LC_MESSAGES, [1] (importlib.abc.SourceLoader method)
LC_MONETARY (in module locale) (importlib.abc.SourcelessFileLoader method)
LC_NUMERIC (in module locale) (importlib.machinery.SourceFileLoader method)
LC_TIME (in module locale) (importlib.machinery.ModuleSpec attribute)
lchflags() (in module os) (in module os)
lchown() (in module os) (pathlib.Path method)
LDCXXSHARED
ldexp() (in module math) (in module)
LDFLAGS
ldgettext() (in module gettext) Loader (class in importlib.abc)
ldngettext() (in module gettext) loader (class)
le() (in module operator) attribute
leading whitespace loader_state (attribute)
leapdays() (in module calendar) LoadError
leaveok() (curses.window method) LoadKey() (in module)
left (filecmp.dircmp attribute) LoadLibrary() (method)
left() (in module turtle) loads() (in module)
left_list (filecmp.dircmp attribute) (in module)
left_only (filecmp.dircmp attribute) (in module)
LEFTSHIFT (in module token) (in module)
LEFTSHIFTEQUAL (in module token) (in module)
len loadTestsFromModule()
     built-in function, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] loadTestsFromName()
len() (built-in function) loadTestsFromNames()
length (xml.dom.NamedNodeMap attribute) loadTestsFromModule()
     (xml.dom.NodeList attribute) loadTestsFromName()
length_hint() (in module operator) loadTestsFromNames()
LESS (in module token)
LESSEQUAL (in module token)
lexical analysis
lexical definitions
lexists() (in module os.path)
lgamma() (in module math)
lgettext() (gettext.GNUTranslations method)
  (gettext.NullTranslations method)
  (in module gettext)
lib2to3 (module)
libc_ver() (in module platform)
library (in module dbm.ndbm)
  (ssl.SSLError attribute)
library_dir_option()
  (distutils.ccompiler.CCompiler method)
library_filename()
  (distutils.ccompiler.CCompiler method)
library_option()  (distutils.ccompiler.CCompiler method)
LibraryLoader (class in ctypes)
license (built-in variable)
LifoQueue (class in asyncio)
  (class in queue)
light-weight processes
limit_denominator()  (fractions.Fraction method)
LimitOverrunError
lin2adpcm() (in module audioop)
lin2alaw() (in module audioop)
lin2lin() (in module audioop)
lin2ulaw() (in module audioop)
line (C function)
line continuation
line joining, [1]
loadTestsFromTestCase() (unittest.TestLoader method)
local (class in threading)
localcontext() (locale (module)
localeconv() (locale (module)
LocaleHTMLCalendar (class in calendar)
LocaleTextCalendar (class in calendar)
localName (xml.dom.Node attribute)
locals() (built-in function)
localtime() (in module time)
Locator (class in xml.sax.xmlreader)
Lock (class in asyncio)
  (class in multiprocessing)
  (class in threading)
lock() (mailbox.Mailbox method)
lock,helde() (in asyncio.Condition method)
locked() _thread.lock method
lockf() (in module fcntl)
line structure

line() (msilib.Dialog method)

line-buffered I/O

line_buffering (io.TextIOWrapper attribute)

line_num (csv.csvreader attribute)

linocache (module)

lineno (ast.AST attribute)

(line_test.DocTest attribute)

(line_test.Example attribute)

(json.JSONDecodeError attribute)

(pyclbr.Class attribute)

(pyclbr.Function attribute)

(re.error attribute)

(shlex.shlex attribute)

(traceback.TracebackException attribute)

(tracemalloc.Filter attribute)

(tracemalloc.Frame attribute)

(xml.parsers.expat.ExpatError attribute)

lineno() (in module fileinput)

LINES, [1], [2], [3], [4]

lines (os.terminal_size attribute)

linesep (email.policy.Policy attribute)

    (in module os)

lineterminator (csv.Dialect attribute)

LineTooLong

link() (distutils.ccompiler.CCompiler method)

    (in module os)

link_executable()

    (distutils.ccompiler.CCompiler method)

link_shared_lib()

    (distutils.ccompiler.CCompiler method)

link_shared_object()

    (distutils.ccompiler.CCompiler method)

linename (tarfile.TarInfo attribute)

logging

    (in module fileinput)

LockType (in module threading)

log() (in module math)

    (in module math)

log10() (decimal.Decimal method)

    (decimal.Decimal method)

log1p() (math.

    (math.

log2() (math.

    (math.

log_date_time_string()

    (http.server.BaseHTTPRequestHandler method)

log_error()

    (http.server.BaseHTTPRequestHandler method)

log_to_stderr()

    (multiprocessing)

logb() (decimal.Decimal method)

    (decimal.Decimal method)

Logger (class in logging)

LoggerAdapter (class in logging)

logging

Errors
list

assignment, target
comprehensions
deletion target
display
empty
expression, [1], [2]
object, [1], [2], [3], [4], [5], [6], [7], [8]
target, [1]
type, operations on

list (built-in class)
List (class in typing)
list (pdb command)

list comprehension
list() (imaplib.IMAP4 method)
    (multiprocessing.managers.SyncManager method)
    (nntplib.NNTP method)
    (poplib.POP3 method)
    (tarfile.TarFile method)
LIST_APPEND (opcode)
list_dialects() (in module csv)
list_folders() (mailbox.Maildir method)
    (mailbox.MH method)
listdir() (in module os)
listen() (asyncore.dispatcher method)
    (in module logging.config)
    (in module turtle)
    (socket.socket method)
Listener (class in multiprocessing.connection)
listMethods()
(xmlrpc.client.ServerProxy.system method)
ListNoteBook (class in tkinter.tix)
listxattr() (in module os)
literal, [1]
literal_eval() (in module ast)
literals
  binary
complex number
floating point
hexadecimal
integer
numeric
octal
LittleEndianStructure (class in ctypes)
ljust() (bytearray method)
  (bytes method)
  (str method)
LK_LOCK (in module msvcrt)
LK_NBLCK (in module msvcrt)
LK_NBRLCK (in module msvcrt)
over mutable statement,
loop control
target
loop() (in module
lower() (bytearray
  (bytes method)
  (str method)
LPAR (in module
lpAttributeList
attribute)
Iru_cache() (in
Iseek() (in mod
Ishift() (in mod
LSQB (in mod
Istat() (in mod
  (pathlib.Pa
Istrip() (bytearray
  (bytes method)
  (str method)
Lsub() (imaplib.
lt() (in module
  (in module
LWPCookieJar
Izma (module)
LZMACompressor
LZMADecompressor
LZMAError
LZMAFile (clas
M (in module re)
mac_ver() (in module platform)
machine() (in module platform)
macpath (module)
macros (netrc.netrc attribute)
MAGIC_NUMBER (in module importlib.util)
MagicMock (class in unittest.mock)
Mailbox (class in mailbox)
mailbox (module)
mailcap (module)
Maildir (class in mailbox)
MaildirMessage (class in mailbox)
mailfrom (smtplib.SMTPChannel attribute)
MailmanProxy (class in smtplib)
main(), [1], [2] (in module py_compile)
mainloop() (in module turtle)
maintype (email.headerregistry.ContentTypeHeader attribute)
major (email.headerregistry.MIMEVersionHeader attribute)
major() (in module os)
make_alternative() (email.message.EmailMessage method)
make_archive() (in module distutils.archive_util)
methods (in module pyclbr.Class)
methodSignature() (xmlrpc.client.ServerProxy.system method)
MethodType (in module types)
MethodWrapperType (in module types)
MH (class in mailbox)
MHMessage (class)
microsecond (datetime.datetime attribute)
MIME
base64 encoding
content type
headers, [1]
quoted-printable encoding
MIMEApplication
e-mail.mime.application
MIMEAudio (class)
MIMEBase (class)
MIMEImage (class)
MIMEMessage
e-mail.mime.message
MIMEApplication
e-mail.mime.multipart
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
MIMEApplication
e-mail.mime.multipart
makefile() (socket method)
     (socket.socket method)
makeLogRecord() (in module logging)
makePickle()
(logging.handlers.SocketHandler method)
makeRecord() (logging.Logger method)
makeSocket()
(logging.handlers.DatagramHandler method)
     (logging.handlers.SocketHandler method)
maketrans() (bytearray static method)
     (bytes static method)
     (str static method)
malloc()
mangle_from_ (email.policy.Compat32 attribute)
     (email.policy.Policy attribute)
mangling
     name, [1]
map (2to3 fixer)
map() (built-in function)
     (concurrent.futures.Executor method)
     (multiprocessing.pool.Pool method)
     (tkinter.ttk.Style method)
MAP_ADD (opcode)
mock_add_spec() (multiprocessing.pool.Pool method)
mock_calls (unittest.mock.Mock attribute)
mock_open() (in module mock)
mkdir() (in module os)
(pathlib.Path method)
mkdirtemp() (in module multiprocessing.pool)
mkfifo() (in module os)
mknod() (in module os)
mkpath() (distutils.ccompiler.CCompiler method)
     (in module distutils.dir_util)
mksalt() (in module crypt)
mkstemp() (in module tempfile)
mktemp() (in module tempfile)
mktime() (in module time)
mktime_tz() (in module email.utils)
mlsd() (ftplib.FTP method)
mmap (class in mmap module)
MMDF (class in mailbox)
MMDFMessage (class in mailbox)
Mock (class in unittest.mock)
mock_add_spec() (method)
mock_calls (unittest.mock.Mock attribute)
mock_open() (in module mock)
mmode (io.FileIO attribute)
missing_compiler_executable() (in module test.support)
MissingSectionHeaderError
 MIXERDEV
mkd() (ftplib.FTP method)
mkdir() (in module os)
mmode (io.FileIO attribute)
MISSING_C_DOC
mapLogRecord()
(logging.handlers.HTTPHandler method)

**mapping**

- object, [1], [2], [3], [4], [5]
- types, operations on
- Mapping (class in collections.abc)
  (class in typing)
- mapping() (msilib.Control method)
- MappingProxyType (class in types)
- MappingView (class in collections.abc)
  (class in typing)
- mapPriority()
  (logging.handlers.SysLogHandler method)
- maps (collections.ChainMap attribute)
- maps() (in module nis)
- marshal (module)

**marshalling**

- objects

**masking**

- operations
- Match (class in typing)
- match() (in module nis)
  (in module re)
  (pathlib.PurePath method)
  (re.Pattern method)
- match_hostname() (in module ssl)
- match_test() (in module test.support)
- match_value() (test.support.Matcher method)
- Matcher (class in test.support)
- matches() (test.support.Matcher method)

**math**

- module, [1]
- math (module)

attribute)
  (tarfile.TarInfo
mode() (in module
  (in module tur
modes
  
file
modf() (in module
modified()
  (urllib.robotparser.
  method)
Modify() (msilib.Vi
  modify() (select.de
  (select.epoll no
  (select.poll m
  (selectors.Base

**module**

__main__, [1]
  _locale
  _thread
  array, [1]
  base64
  bdb
  binhex
  builtins, [1], [2]
  cmd
  copy
  crypt
dbm.gnu, [1]
  dbm.ndbm, [1]
erro
extension
glob
imp
importing
matmul() (in module operator)
matrix multiplication
max
    built-in function
max (datetime.date attribute)
    (datetime.datetime attribute)
    (datetime.time attribute)
    (datetime.timedelta attribute)
max() (built-in function)
    (decimal.Context method)
    (decimal.Decimal method)
    (in module audioop)
max_count
    (email.headerregistry.BaseHeader attribute)
MAX_EMAX (in module decimal)
MAX_INTERPOLATION_DEPTH (in module configparser)
max_line_length
    (email.policy.Policy attribute)
max_lines
    (textwrap.TextWrapper attribute)
max_mag()
    (decimal.Context method)
    (decimal.Decimal method)
max_memuse
    (in module test.support)
MAX_PREC (in module decimal)
max_prefixlen
    (ipaddress.IPv4Address attribute)
    (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Address attribute)
    (ipaddress.IPv6Network attribute)
MAX_Py_ssize_t (in module test.support)
maxarray
    (reprlib.Repr attribute)
maxdeque
    (reprlib.Repr attribute)
io
json
math, [1]
namespace
object, [1], [2]
os
pickle, [1], [2], pty
pwd
pyexpat
re, [1]
search path,
[7]
shelve
signal, [1]
sitecustomize
socket
stat
string
struct
sys, [1], [2], [3]
types
urllib.request
usercustomize
uu
module (pyclbr.Class attribute)
    (pyclbr.Function attribute)
module spec, [1]
module_for_loader
importlib.util
module_from_spec
importlib.util
module_repr()
method
maxdict (reprlib.Repr attribute)
maxDiff (unittest.TestCase attribute)
maxfrozenset (reprlib.Repr attribute)
MAXIMUM_SUPPORTED (ssl.TLSVersion attribute)
maximum_version (ssl.SSLContext attribute)
maxlen (collections.deque attribute)
maxlevel (reprlib.Repr attribute)
maxlist (reprlib.Repr attribute)
maxlong (reprlib.Repr attribute)
maxother (reprlib.Repr attribute)
maxpp() (in module audioop)
maxset (reprlib.Repr attribute)
maxsize (asyncio.Queue attribute)
maxstring (reprlib.Repr attribute)
maxtuple (reprlib.Repr attribute)
maxunicode (in module sys)
MAXYEAR (in module datetime)
MB_ICONASTERISK (in module winsound)
MB_ICONEXCLAMATION (in module winsound)
MB_ICONHAND (in module winsound)
MB_ICONQUESTION (in module winsound)
MB_OK (in module winsound)
mbox (class in mailbox)
mboxMessage (class in mailbox)
mean() (in module statistics)
median() (in module statistics)
median_grouped() (in module statistics)
median_high() (in module statistics)
ModuleFinder (class in modulefinder)
modulefinder (module)
ModuleInfo (class in modulefinder)
ModuleNotFoundError
modules (in module modulefinder)
modules_cleanup() (in module test.support)
modules_setup() (in module test.support)
ModuleSpec (class in importlib.machinery)
ModuleType (class in module types)
modulo
monotonic() (in module time)
monotonic_ns() (in module time)
month (datetime.date attribute)
month() (in module datetime)
month_abbr (in module calendar)
month_name (in module calendar)
monthcalendar() (in module calendar)
monthdays2calendar() (calendar.Calendar method)
monthdayscalendar() (calendar.Calendar method)
monthrange() (in module calendar)
Morsel (class in http.cookies)
most_common() (collections.Counter method)
mouseinterval() (in module curses)
mousemask() (in module curses)
move() (curses.panel method)
median_low() (in module statistics)

MemberDescriptorType (in module types)

member

test

memmove() (in module ctypes)

MemoryBIO (class in ssl)

MemoryError

MemoryHandler (class in logging.handlers)

memoryview

   object, [1]

memoryview (built-in class)

memset() (in module ctypes)

merge() (in module heapq)

Message (class in email.message)

   (class in mailbox)

message digest, MD5

message_factory (email.policy.Policy attribute)

message_from_binary_file() (in module email)

message_from_bytes() (in module email)

message_from_file() (in module email)

message_from_string() (in module email)

MessageBeep() (in module winsound)

MessageClass

   (http.server.BaseHTTPRequestHandler attribute)

MessageError

MessageParseError

messages (in module xml.parsers.expat.errors)

meta

   hooks

curses.window

   (in module shutil)

mmap.mmap

   (tkinter.ttk.Treeview method)

move_file() (distutils.ccompiler.CCompiler method)

MozillaCookieJar

MRO

mro() (class method)

msg (http.client.HTTPResponse attribute)

msg() (telnetlib.Telnet method)

msilib (module)

msvcrt (module)

mt_interact() (telnetlib.Telnet method)

mtime (gzip.GzipFile attribute)

mtime() (tarfile.TarInfo attribute)

mul() (in module audioop)

mul() (in module operator)

MultiCall (class in xmlrpc.client)

MULTILINE (in module re)

MultipartConversionError

multiplication

multiply() (decimal.Context method)
meta hooks

**meta path finder**

meta() (in module curses)
meta_path (in module sys)
metaclass, [1]
    (2to3 fixer)
metaclass hint
MetaPathFinder (class in importlib.abc)
metavar (optparse.Option attribute)
MetavarTypeHelpFormatter (class in argpargc)
Meter (class in tkinter.tix)
METH_CLASS (built-in variable)
METH_COEXIST (built-in variable)
METH_KEYWORDS (built-in variable)
METH_NOARGS (built-in variable)
METH_O (built-in variable)
METH_STATIC (built-in variable)
METH_VARARGS (built-in variable)
method
    built-in
call
    object, [1], [2], [3], [4], [5]
    user-defined
method (urllib.request.Request attribute)

**method resolution order**

METHOD_BLOWFISH (in module crypt)
method_calls (unittest.mock.Mock attribute)
METHOD_CRYPT (in module crypt)
METHOD_MD5 (in module crypt)
METHOD_SHA256 (in module crypt)
METHOD_SHA512 (in module crypt)
methodattrs (2to3 fixer)

**mutable**

object, [1], [2]
sequence type
mutable object
mutable sequence
loop over
object
MutableMapping (class in collections.abc)
    (class in typing)
MutableSequence (class in collections.abc)
    (class in typing)
MutableSet (class in typing)
    mvderwin() (curses.window method)
mvwin() (curses.window method)
myrights() (imaplib.IMAP4 method)
methodcaller() (in module operator)
MethodDescriptorType (in module types)
methodHelp()
(xmlrpc.client.ServerProxy.system method)
methods
    bytearray
    bytes
    string
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ngettext()</td>
<td>gettext.GNUTranslations method</td>
</tr>
<tr>
<td>ngettext()</td>
<td>gettext.NullTranslations method</td>
</tr>
<tr>
<td>nice()</td>
<td>os (module)</td>
</tr>
<tr>
<td>nis()</td>
<td>nntplib (class)</td>
</tr>
<tr>
<td>nl_langinfo()</td>
<td>(contextvars.ContextVar attribute)</td>
</tr>
<tr>
<td>nlargest()</td>
<td>(doctest.DocTest attribute)</td>
</tr>
<tr>
<td>nlst()</td>
<td>(email.headerregistry.BaseHeader attribute)</td>
</tr>
<tr>
<td>NNTP protocol</td>
<td>(importlib.abc.FileLoader attribute)</td>
</tr>
<tr>
<td>NNTP_SSL</td>
<td>(importlib.machinery.ExtensionFileLoader attribute)</td>
</tr>
<tr>
<td>NNTPDataError</td>
<td>(importlib.machinery.ModuleSpec attribute)</td>
</tr>
<tr>
<td>NNTPPermanentError</td>
<td>(importlib.machinery.SourceFileLoader attribute)</td>
</tr>
<tr>
<td>NNTPReplyError</td>
<td>(importlib.machinery.SourcelessFileLoader attribute)</td>
</tr>
<tr>
<td>no_proxy</td>
<td>os (module)</td>
</tr>
<tr>
<td>no_tracing()</td>
<td>test.support</td>
</tr>
<tr>
<td>no_type_check()</td>
<td></td>
</tr>
</tbody>
</table>

N

N_TOKENS (in module token)
n_waiting (threading.Barrier attribute)
name, [1], [2]
  binding, [1], [2], [3], [4], [5], [6]
  binding, global
class
  function, [1]
  mangling, [1]
  rebinding
  unbinding
name (codecs.CodecInfo attribute)
  (contextvars.ContextVar attribute)
  (doctest.DocTest attribute)
  (email.headerregistry.BaseHeader attribute)
  (hashlib.hash attribute)
  (hmac.HMAC attribute)
  (http.cookiejar.Cookie attribute)
  (importlib.abc.FileLoader attribute)
  (importlib.machinery.ExtensionFileLoader attribute)
  (importlib.machinery.ModuleSpec attribute)
  (importlib.machinery.SourceFileLoader attribute)
  (importlib.machinery.SourcelessFileLoader attribute)
  (in module os)
NAME (in module token)
name (inspect.Parameter attribute)
  (io.FileIO attribute)
  (multiprocessing.Process attribute)
named tuple
NamedTemporaryFile() (in module tempfile)
NamedTuple (class in typing)
namedtuple() (in module collections)
NameError exception
NameError (built-in exception)
namelist() (zipfile.ZipFile method)
namereplace_errors() (in module codecs)

names
    private
namespace, [1]
    global
    module
    package
Namespace (class in argparse)
    (class in multiprocessing.managers)

namespace package
namespace() (imaplib.IMAP4 method)
Namespace()
    (multiprocessing.managers.SyncManager
no_type_check_decorator() (in module typing)
nocbreak() (in module curses)
NoDataAllowedErr
node() (in module platform)
nodelay() method
nodeName attribute
NodeTransformer
nodeType attribute
nodeValue attribute
NodeVisitor
noecho() in module
NOEXPR in module
NoModificatio
nonblock() (ossaudiodev.
method)
NonCallableV
in unittest.mock
NonCallableV
unittest.mock)
None
        object, [1.
None (Built-in
    (built-in v
nonl() (in module
nonlocal
        statement
nonzero (2to3
method)
NAMESPACE_DNS (in module uuid)
NAMESPACE_OID (in module uuid)
NAMESPACE_URL (in module uuid)
NAMESPACE_X500 (in module uuid)
NamespaceErr
namespaceURI (xml.dom.Node attribute)
NaN
nan (in module cmath)
  (in module math)
nanj (in module cmath)
NannyNag
napms() (in module curses)
nargs (optparse.Option attribute)
nbytes (memoryview attribute)
ndiff() (in module difflib)
ndim (memoryview attribute)
ne (2to3 fixer)
ne() (in module operator)
needs_input (bz2.BZ2Decompressor attribute)
  (lzma.LZMADecompressor attribute)
neg() (in module operator)
negation
netmask (ipaddress.IPv4Network attribute)
  (ipaddress.IPv6Network attribute)
NetmaskValueError
netrc (class in netrc)
  (module)
NetrcParseError
netscape (http.cookiejar.CookiePolicy attribute)
network (ipaddress.IPv4Interface attribute)
  (ipaddress.IPv6Interface attribute)
Network News Transfer Protocol
noop() (imaplib.IMAP4 method)
  (poplib.POP3 method)
NoOptionError
NOP (opcode)
nofqiflush() (in
noraw() (in
NoReturn (in
NORMAL_PRIORITY_CLASS (in module subprocess)
normalize() (decimal.
  (in module
 normalize() (decimal.
  (in module
 NORMALIZE_ (in module
 NORMALIZE_ (in module
 normal-case random
 normcase() (in
 normpath() (in
 NoSectionError
 NoSuchMailbox
 not
 not
 not
 operator, operator, operator, operator,
 not in
 not in
 NotADirectory
 notation
 notationDecl() (xml.sax.handler.DTDHandler method)
network_address (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Network attribute)
new() (in module hashlib)
    (in module hmac)
new-style class
new_alignment() (formatter.writer method)
new_child() (collections.ChainMap method)
new_class() (in module types)
new_compiler() (in module distutils.ccompiler)
new_event_loop()
    (asyncio.AbstractEventLoopPolicy method)
    (in module asyncio)
new_font() (formatter.writer method)
new_margin() (formatter.writer method)
new_module() (in module imp)
new_panel() (in module curses.panel)
new_spacing() (formatter.writer method)
new_styles() (formatter.writer method)
newer() (in module distutils.dep_util)
newer_group() (in module distutils.dep_util)
newer_pairwise() (in module distutils.dep_util)
newgroups() (nntplib.NNTP method)
NEWLINE (in module token)
NEWLINE token, [1]
newlines (io.TextIOBase attribute)
newnews() (nntplib.NNTP method)
newpad() (in module curses)
NewType() (in module typing)
newwin() (in module curses)
next (2to3 fixer)
    (pdb command)
next() (built-in function)
    (nntplib.NNTP method)
(tarfile.TarFile method)
(tkinter.ttk.Treeview method)
next_minus() (decimal.Context method)
   (decimal.Decimal method)
next_plus() (decimal.Context method)
   (decimal.Decimal method)
next_toward() (decimal.Context method)
   (decimal.Decimal method)
nexfile() (in module fileinput)
nexkey() (dbm.gnu.gdbm method)
nexSibling (xml.dom.Node attribute)

method)
NSIG (in mod
nsmallest() (ir
NT_OFFSET
NTEventLog-
logging.handle
ntohl() (in mo
ntohs() (in mo
mtransfercmd( method)
nul
    operation
nullcontext()
contextlib)
NullFormatter
formatt
NullHandler (c
NullImporter ( Nul
NullTranslati
gettext)
NullWriter (cl
number
   complex
   floating p
Number (cl: NUMBER (in
number_class
   (decimal. Co
   (decimal. C
numbers (module)
  numerator (numbers attribute)
  denominator (numbers attribute)

numeric
  numeric literal
  numeric object, [1]
  numeric literal numeric()
  numeric literal unicodedata

Numerical Python
  numinput() (in module turtle)
  numliterals (2)
O_APPEND (in module os)
O_ASYNC (in module os)
O_BINARY (in module os)
O_CLOEXEC (in module os)
O_CREAT (in module os)
O_DIRECT (in module os)
O_DIRECTORY (in module os)
O_DSYNC (in module os)
O_EXCL (in module os)
O_EXLOCK (in module os)
O_NDELAY (in module os)
O_NOATIME (in module os)
O_NOCTTY (in module os)
O_NOFOLLOW (in module os)
O_NOINHERIT (in module os)
O_NONBLOCK (in module os)
O_PATH (in module os)
O_RANDOM (in module os)
O_RDONLY (in module os)
O_RDWR (in module os)
O_RSYNC (in module os)
O_SEQUENTIAL (in module os)
O_SHLOCK (in module os)
O_SHORT_LIVED (in module os)
O_SYNC (in module os)
O_TEMPORARY (in module os)
O_TEXT (in module os)
O_TMPFILE (in module os)
O_TRUNC (in module os)
O_WRONLY (in module os)
obj (memoryview attribute)
object, [1]

open_binary()
importlib.resources)
open_connection()
asyncio)
open_new() (in modu
webbrowser.co
open_new_tab()
webbrowser)
open_osfhandle() (in
open_resource() (importlib.abc.Resou
method)
open_text() (importlib.resources)
open_unix_connection
asncio)
open_unknown()
(urllib.request.URLop
open_urlresource()
test.support)
OpenDatabase() (in
OpenerDirector
urllib.request)
openfp() (in module s
(in module wave)
OpenKey() (in modul
OpenKeyEx() (in mod
openlog() (in module
openmixer() (in mod
openpty() (in module
Boolean, [1]  
Capsule  
Ellipsis  
None, [1], [2]  
NotImplemented  
asynchronous-generator  
built-in function, [1]  
built-in method, [1]  
bytestring, [1], [2], [3]  
bytestring, [1], [2]  
callable, [1]  
class, [1], [2]  
class instance, [1], [2]  
code, [1], [2], [3]  
complex  
complex number, [1]  
deallocation  
dictionary, [1], [2], [3], [4], [5], [6], [7]  
file, [1]  
finalization  
floating point, [1], [2]  
frame  
frozenset, [1]  
function, [1], [2], [3], [4], [5]  
generator, [1], [2]  
immutable, [1], [2]  
immutable sequence  
instance, [1], [2]  
instancemethod  
integer, [1], [2]  
io.StringIO  
list, [1], [2], [3], [4], [5], [6], [7], [8]  
long integer  

OpenSSL  
(uset in module '  
(uset in module '  
OPENSSL_VERSION  
OPENSSL_VERSION  
module ssl)  
OPENSSL_VERSION  
module ssl)  
OpenView() (msilib.  
operation  
operation

Boolean  
binary arithmetic  
binary bitwise  
concatenation  
nul1, [1]  
power  
repetition  
shifting  
slice  
subscript  
uminary arithmetic  
uminary bitwise  
OperationalError  
operations  
operations  
on  
odes  
dictionary type  
type  
type  

type  
numeric types
mapping, [1, 2, 3, 4, 5]
memoryview, [1]
method, [1, 2, 3, 4, 5]
module, [1, 2]
mutable, [1, 2]
mutable sequence
numeric, [1, 2, 3, 4]
range
sequence, [1, 2, 3, 4, 5, 6, 7, 8]
set, [1, 2, 3]
set type
slice
socket
string, [1, 2]
traceback, [1, 2, 3, 4]
tuple, [1, 2, 3, 4, 5, 6]
type, [1, 2]
user-defined function, [1, 2]
user-defined method
object (built-in class)
  (UnicodeError attribute)
object.__slots__ (built-in variable)
object_filenames()
(distutils.ccompiler.CCompiler method)
objects
  comparing
  flattening
  marshalling
  persistent
  pickling
  serializing
obufcount()
(ossaudiodev.oss_audio_device
sequence types,
operator
!=, [1]
% (percent), [1]
& (ampersand),
* (asterisk), [1]
**, [1]
+ (plus), [1, 2]
- (minus), [1, 2]
/ (slash), [1]
//, [1]
< (less), [1]
<<, [1]
<=, [1]
==, [1]
>, [1]
>=, [1]
>>, [1]
@ (at)
^ (caret), [1]
and, [1, 2]
comparison
in, [1, 2]
is, [1]
is not, [1]
not, [1]
not in, [1, 2]
or, [1, 2]
overloading
precedence
ternary
| (vertical bar), [:]
~ (tilde), [1]
operator (2to3 fixer)
method)
obuffree()
(ossaudiodev.oss_audio_device
method)
oct() (built-in function)

octal
    literals
    octal literal
octdigits (in module string)
offset (traceback.TracebackException
attribute)
    (xml.parsers.expat.ExpatError
attribute)
OK (in module curses)
old_value
(contextvars.contextvars.Token.Token
attribute)
OleDLL (class in ctypes)
onclick() (in module turtle), [1]
ondrag() (in module turtle)
onecmd() (cmd.Cmd method)
onkey() (in module turtle)
onkeypress() (in module turtle)
onkeyrelease() (in module turtle)
onrelease() (in module turtle)
screensclick() (in module turtle)
tonimer() (in module turtle)
OP (in module token)
OP_ALL (in module ssl)
OP_CIPHER_SERVER_PREFERENCE
(in module ssl)
OP_ENABLE_MIDDLEBOX_COMPAT
(in module ssl)
OP_NO_COMPRESSION (in module
operators
opmap (in module dis)
opname (in module dis)
opim_args_from_int
attribute)
optimize() (in module
OPTIMIZED_BYTECODE
attribute)
Optional (in module
OptionGroup (class in
OptionMenu (class in
OptionParser (class in
Options (class in ssl)
options (doctest.Example
ssl.SSLContext
options() (configparser
method)
optionxform() (configparser.ConfigParser
[1]
optparse (module)

or
    bitwise
    exclusive
    inclusive
    operator, [1], [2]
or_() (in module operator)
ord
    built-in function
ord() (built-in function)
order
    evaluation
ordered_attributes
OP_NO_RENEGOTIATION (in module ssl)
OP_NO_SSLv2 (in module ssl)
OP_NO_SSLv3 (in module ssl)
OP_NO_TICKET (in module ssl)
OP_NO_TLSv1 (in module ssl)
OP_NO_TLSv1_1 (in module ssl)
OP_NO_TLSv1_2 (in module ssl)
OP_NO_TLSv1_3 (in module ssl)
OP_SINGLE_DH_USE (in module ssl)
OP_SINGLE_ECDH_USE (in module ssl)
open built-in function, [1]
open() (built-in function)
(distutils.text_file.TextFile method)
(imaplib.IMAP4 method)
(in module aifc)
(in module bz2)
(in module codecs)
(in module dbm)
(in module dbm.dumb)
(in module dbm.gnu)
(in module dbm.ndbm)
(in module gzip)
(in module io)
(in module lzma)
(in module os)
(in module ossaudiodev)
(in module shelf)
(in module sunau)
(in module tarfile)
(in module tokenize)

(OrderedDict (class in collections)
(OrderedDict (class in typing)
(origin (importlib.machinery. attribute)
(origin_req_host (urllib.request.Request.Reque
(origin_server (wsgiref.handlers.Ba attribute)
(os (module)
(os (module)
(os.environ (wsgiref.handlers.Ba attribute)
(OSError
(ossaudiodev (module)
(OSSAudioError
outfile

(json.tool comma
output

(output (subprocess.CalledP attribute)

(output (subprocess.Tim attribute)

(output (unittest.TestCase	attribute)

(output()) (http.co	method)
(in module wave)
(in module webbrowser)
(pathlib.Path method)
(pipes.Template method)
(tarfile.TarFile class method)
(telnetlib.Telnet method)
(urllib.request.OpenerDirector method)
(urllib.request.URLopener method)
(webbrowser.controller method)
(zipfile.ZipFile method)

(http.cookies.Morsel method)

(output_charset (email.charset.Charset attribute))
(output_charset() (gettext.NullTranslations method)
(output_codec (email.charset.Charset attribute))
(output_difference() (doctest.OutputChecker class)
(OutputString() (http.cookies.Morsel method)
over() (nntplib.NNTP method)
Overflow (class in decimal)
OverflowError (built-in exception)
[4], [5]
overlaps() (ipaddress.IPv4Network method)
/ipaddress.IPv6Network method)
overlay() (curses.window method)
overloading
operator
overwrite() (curses.window method)
owner() (pathlib.Path method)
p (pdb command)
P_ALL (in module os)
P_DETACH (in module os)
P_NOWAIT (in module os)
P_NOWAITO (in module os)
P_OVERLAY (in module os)
P_PGID (in module os)
P_PID (in module os)
P_WAIT (in module os)
pack() (in module struct)
    (mailbox.MH method)
    (struct.Struct method)
pack_array() (xdrlib.Packer method)
pack_bytes() (xdrlib.Packer method)
pack_double() (xdrlib.Packer method)
pack_farray() (xdrlib.Packer method)
pack_float() (xdrlib.Packer method)
pack_fopaque() (xdrlib.Packer method)
pack_fstring() (xdrlib.Packer method)
pack_into() (in module struct)
    (struct.Struct method)
pack_list() (xdrlib.Packer method)
pack_opaque() (xdrlib.Packer method)
pack_string() (xdrlib.Packer method)
package, [1], [2]
    namespace
    portion
    regular
Package (in module importlib.resources)
package variable
    __all__
packed (ipaddress.IPv4Address attribute)
Packer (class in xdrlib)

packing
  binary data

packing (widgets)

PAGER

pair_content() (in module curses)
pair_number() (in module curses)

PanedWindow (class in tkinter.tix)

parameter
  call semantics
  difference from argument
  function definition
  value, default

Parameter (class in inspect)

ParameterizedMIMEHeader (class in email.headerregistry)

parameters (inspect.Signature attribute)

params

(email.headerregistry.ParameterizedMIMEHeader attribute)

pardir (in module os)

paren (2to3 fixer)

parent (importlib.machinery.ModuleSpec attribute)
  (pyclbr.Class attribute)
  (pyclbr.Function attribute)
  (urllib.request.BaseHandler attribute)

parent() (tkinter.ttk.Treeview method)

parenthesized form

parentNode (xml.dom.Node attribute)

parents (collections.ChainMap attribute)

paretovariate() (in module random)

parse() (doctest.DocTestParser method)
  (email.parser.BytesParser method)
ParseError (class in xml.etree.ElementTree)
ParseFile() (xml.parsers.expat.xmlparser method)
ParseFlags() (in module imaplib)
parser
Parser (class in email.parser)
parser (module)
ParserCreate() (in module xml.parsers.expat)
ParserError
ParseResult (class in urllib.parse)
ParseResultBytes (class in urllib.parse)
parsestr() (email.parser.Parser method)
parseString() (in module xml.dom.minidom)
    (in module xml.dom.pulldom)
    (in module xml.sax)
parsing
    Python source code
    URL
ParsingError
partial (asyncio.IncompleteReadError attribute)
partial() (imaplib.IMAP4 method)
    (in module functools)
partialmethod (class in functools)
parties (threading.Barrier attribute)
partition() (bytearray method)
    (bytes method)
    (str method)
pass
    statement
pass_() (poplib.POP3 method)
Paste
patch() (in module test.support)
    (in module unittest.mock)
patch.dict() (in module unittest.mock)
patch.multiple() (in module unittest.mock)
patch.object() (in module unittest.mock)
patch.stopall() (in module unittest.mock)
PATH, [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34]
path
  configuration file
  hooks
  module search, [1], [2], [3], [4], [5], [6], [7]
  operations, [1]
Path (class in pathlib)
path (http.cookiejar.Cookie attribute)
  (http.server.BaseHTTPRequestHandler attribute)
  (importlib.abc.FileLoader attribute)
  (importlib.machinery.ExtensionFileLoader attribute)
  (importlib.machinery.FileFinder attribute)
  (importlib.machinery.SourceFileLoader attribute)
  (importlib.machinery.SourcelessFileLoader attribute)
  (in module sys), [1], [2], [3], [4]
  (os.DirEntry attribute)
path based finder, [1]
Path browser
path entry
path entry finder
path entry hook
path hooks
path() (in module importlib.resources)
path-like object
path_hook() (importlib.machinery.FileFinder class method)
path_hooks (in module sys)
path_importer_cache (in module sys)
path_mtime() (importlib.abc.SourceLoader method)
path_return_ok() (http.cookiejar.CookiePolicy method)
path_stats() (importlib.abc.SourceLoader method)
pathconf() (in module os)
pathconf_names (in module os)
PathEntryFinder (class in importlib.abc)
PATH_EXTRACT, [1]
PathFinder (class in importlib.machinery)
pathlib (module)
PathLike (class in os)
pathname2url() (in module urllib.request)
pathsep (in module os)
Pattern (class in typing)
pattern (re.error attribute)
  (re.Pattern attribute)
pause() (in module signal)
pause_reading() (asyncio.ReadTransport method)
pause_writing() (asyncio.BaseProtocol method)
PAX_FORMAT (in module tarfile)
pax_headers (tarfile.TarFile attribute)
  (tarfile.TarInfo attribute)
pbkdf2_hmac() (in module hashlib)
pd() (in module turtle)
Pdb (class in pdb), [1]
pdb (module)
peek() (bz2.BZ2File method)
  (gzip.GzipFile method)
  (io.BufferedReader method)
  (lzma.LZMAFile method)
  (weakref.finalize method)
peer (smtpd.SMTPChannel attribute)  
PEM_cert_to_DER_cert() (in module ssl)  
pen() (in module turtle)  
pencolor() (in module turtle)  
pending (ssl.MemoryBIO attribute)  
pending() (ssl.SSLSocket method)  
PendingDeprecationWarning  
pendown() (in module turtle)  
pensize() (in module turtle)  
penup() (in module turtle)  
PEP  
PERCENT (in module token)  
PERCENTEQUAL (in module token)  
perf_counter() (in module time)  
perf_counter_ns() (in module time)  
Performance  
PermissionError  
permutations() (in module itertools)  
Persist() (msilib.SummaryInformation method)  
persistence  
persistent  
    objects  
persistent_id (pickle protocol)  
persistent_id() (pickle.Pickler method)  
persistent_load (pickle protocol)  
persistent_load() (pickle.Unpickler method)  
PF_CAN (in module socket)  
PF_PACKET (in module socket)  
PF_RDS (in module socket)  
pformat() (in module pprint)  
    (pprint.PrettyPrinter method)  
PGO (in module test.support)  
phase() (in module cmath)  
Philbrick, Geoff
physical line, [1], [2]
pi (in module cmath)
    (in module math)
pickle
    module, [1], [2], [3]
pickle (module)
pickle() (in module copyreg)
PickleError
Pickler (class in pickle)
pickletools (module)
pickletools command line option
    -a, --annotate
    -l, --indentlevel=<num>
    -m, --memo
    -o, --output=<file>
    -p, --preamble=<preamble>
pickling
    objects
PicklingError
pid (asyncio.asyncio.subprocess.Process attribute)
    (multiprocessing.Process attribute)
    (subprocess.Popen attribute)
PIPE (in module subprocess)
Pipe() (in module multiprocessing)
pipe() (in module os)
pipe2() (in module os)
PIPE_BUF (in module select)
pipe_connection_lost() (asyncio.SubprocessProtocol method)
pipe_data_received() (asyncio.SubprocessProtocol method)
PIPE_MAX_SIZE (in module test.support)
pipes (module)
PKG_DIRECTORY (in module imp)
pkgutil (module)
placeholder (textwrap.TextWrapper attribute)
PLAT
platform (in module sys), [1]
  (module)
platform() (in module platform)
PlaySound() (in module winsound)
plist
  file
plistlib (module)
plock() (in module os)
plus
PLUS (in module token)
plus() (decimal.Context method)
PLUSEQUAL (in module token)
pm() (in module pdb)
POINTER() (in module ctypes)
p POINTER () (in module ctypes)
polar() (in module cmath)
Policy (class in email.policy)
poll() (in module select)
  (multiprocessing.connection.Connection method)
  (select.devpoll method)
  (select.epoll method)
  (select.poll method)
  (subprocess.Popen method)
PollSelector (class in selectors)
Pool (class in multiprocessing.pool)
pop() (array.array method)
  (collections.deque method)
  (dict method)
  (frozenset method)
  (mailbox.Mailbox method)
(sequence method)

POP3

  protocol

POP3 (class in poplib)

POP3_SSL (class in poplib)

pop_alignment() (formatter.formatter method)

pop_all() (contextlib.ExitStack method)

POP_BLOCK (opcode)

POP_EXCEPT (opcode)

pop_font() (formatter.formatter method)

POP_JUMP_IF_FALSE (opcode)

POP_JUMP_IF_TRUE (opcode)

pop_margin() (formatter.formatter method)

pop_source() (shlex.shlex method)

pop_style() (formatter.formatter method)

POP_TOP (opcode)

Popen (class in subprocess)

popen() (in module os), [1], [2]

  (in module platform)

popitem() (collections.OrderedDict method)

  (dict method)

  (mailbox.Mailbox method)

popleft() (collections.deque method)

poplib (module)

PopupMenu (class in tkinter.tix)

port (http.cookiejar.Cookie attribute)

port_specified (http.cookiejar.Cookie attribute)

portion

  package

pos (json.JSONDecodeError attribute)

  (re.Match attribute)

  (re.error attribute)

pos() (in module operator)

  (in module turtle)
position (xml.etree.ElementTree.ParseError attribute)  
position() (in module turtle)  
**positional argument**  
POSIX  
  I/O control  
  threads  
posix (module)  
POSIX_FADV_DONTNEED (in module os)  
POSIX_FADV_NOREUSE (in module os)  
POSIX_FADV_NORMAL (in module os)  
POSIX_FADV_RANDOM (in module os)  
POSIX_FADV_SEQUENTIAL (in module os)  
POSIX_FADV_WILLNEED (in module os)  
posix_fadvise() (in module os)  
posix_fallocate() (in module os)  
POSIXLY_CORRECT  
PosixPath (class in pathlib)  
post() (ntplib.NNTP method)  
  (ossaudiodev.oss_audio_device method)  
post_handshake_auth (ssl.SSLContext attribute)  
post_mortem() (in module pdb)  
post_setup() (venv.EnvBuilder method)  
postcmd() (cmd.Cmd method)  
postloop() (cmd.Cmd method)  
pow  
  built-in function, [1], [2], [3], [4], [5]  
pow() (built-in function)  
  (in module math)  
  (in module operator)  
power  
  operation  
power() (decimal.Context method)  
pp (pdb command)  
pprint (module)
pprint() (in module pprint)  PyObj
    (pprint.PrettyPrinter method)  PyObj
prcal() (in module calendar)  PyObj
pread() (in module os)  PyObj
preadv() (in module os)  PyObj
preamble (email.message.EmailMessage attribute)  PyObj
    (email.message.Message attribute)  PyObj
precedence  PyObj
    operator  PyObj
precmd() (cmd.Cmd method)  PyObj
prefix, [1], [2], [3]  PyObj
PREFIX (in module distutils.sysconfig)  PyObj
prefix (in module sys)  PyObj
    (xml.dom.Attr attribute)  PyObj
    (xml.dom.Node attribute)  PyObj
    (zipimport.zipimporter attribute)  PyObj
PREFIXES (in module site)  PyObj
prefixlen (ipaddress.IPv4Network attribute)  PyObj
    (ipaddress.IPv6Network attribute)  PyObj
preloop() (cmd.Cmd method)  PyObj
prepare() (logging.handlers.QueueHandler method)  PyObj
    (logging.handlers.QueueListener method)  PyObj
prepare_class() (in module types)  PyObj
prepare_input_source() (in module xml.sax.saxutils)  PyObj
prepend() (pipes.Template method)  PyObj
preprocess() (distutils.ccompiler.CCompiler method)  PyObj
PrettyPrinter (class in pprint)  PyObj
prev() (tkinter.ttk.Treeview method)  PyObj
previousSibling (xml.dom.Node attribute)  PyObj
primary  PyObj
print  PyObj
    built-in function  PyObj
print (2to3 fixer)  PyObj
print() (built-in function)  PyObj
__str__() (object method)
print_callees() (pstats.Stats method)
print_callers() (pstats.Stats method)
print_directory() (in module cgi)
print_environ() (in module cgi)
print_environ_usage() (in module cgi)
print_exc() (in module traceback)
(timeit.Timer method)
print_exception() (in module traceback)
PRINT_EXPR (opcode)
print_form() (in module cgi)
print_help() (argparse.ArgumentParser method)
print_last() (in module traceback)
print_stack() (asyncio.Task method)
(in module traceback)
print_stats() (profile.Profile method)
(pstats.Stats method)
print_tb() (in module traceback)
print_usage() (argparse.ArgumentParser method)
(optparse.OptionParser method)
print_version() (optparse.OptionParser method)
printable (in module string)
printdir() (zipfile.ZipFile method)
printf-style formatting, [1]
PRIOR_PGRP (in module os)
PRIOR_PROCESS (in module os)
PRIOR_USER (in module os)
PriorityQueue (class in asyncio)
(class in queue)
private

names
prlimit() (in module resource)
prmmonth() (calendar.TextCalendar method)
(in module calendar)
ProactorEventLoop (class in asyncio)  

procedure
  call

process
  group, [1]
  id
  id of parent
  killing, [1]
  scheduling priority, [1]
  signalling, [1]

Process (class in multiprocessing)
  process() (logging.LoggerAdapter method)
  process_exited() (asyncio.SubprocessProtocol method)
  process_message() (smtpd.SMTPServer method)
  process_request() (socketserver.BaseServer method)
  process_time() (in module time)
  process_time_ns() (in module time)
  process_tokens() (in module tabnanny)
  ProcessError
  processes, light-weight
  ProcessingInstruction() (in module xml.etree.ElementTree)
  processingInstruction()
    (xml.sax.handler.ContentHandler method)
  ProcessingInstructionHandler()
    (xml.parsers.expat.xmlparser method)
  ProcessLookupError
  processor time, [1], [2]
  processor() (in module platform)
  ProcessPoolExecutor (class in concurrent.futures)
  product() (in module itertools)
  Profile (class in profile)

PyOS_vsnprintf (C function)
PyParser_SimpleParseFile (C function)
PyParser_SimpleParseFileFlags (C function)
PyParser_SimpleParseString (C function)
PyParser_SimpleParseStringFlags (C function)
PyParser_SimpleParseStringFlagsFilename (C function)
PyPI (see Python Package Index (PyPI))
PyProperty_Type (C variable)
PyRun_AnyFile (C function)
PyRun_AnyFileEx (C function)
PyRun_AnyFileExFlags (C function)
PyRun_AnyFileFlags (C function)
PyRun_File (C function)
PyRun_FileEx (C function)
PyRun_FileExFlags (C function)
PyRun_FileFlags (C function)
PyRun_InteractiveLoop (C function)
PyRun_InteractiveLoopFlags (C function)
PyRun_InteractiveOne (C function)
PyRun_InteractiveOneFlags (C function)
PyRun_SimpleFile (C function)
PyRun_SimpleFileEx (C function)
PyRun_SimpleFileExFlags (C function)
PyRun_SimpleString (C function)
PyRun_SimpleStringFlags (C function)
PyRun_String (C function)
PyRun_StringFlags (C function)
PySeqIter_Check (C function)
PySeqIter_New (C function)
PySeqIter_Type (C variable)
PySequence_Check (C function)
profile (module)
profile function, [1], [2]
profiler, [1]
profiling, deterministic
program
ProgrammingError
ProgressBar (class in tkinter.ttk)
prompt (cmd.Cmd attribute)
prompt_user_passwd()
(urlib.request.FancyURLopener method)
prompts, interpreter
propagate (logging.Logger attribute)
property (built-in class)
property list
property_declaration_handler (in module xml.sax.handler)
property_dom_node (in module xml.sax.handler)
property_lexical_handler (in module xml.sax.handler)
property_xml_string (in module xml.sax.handler)
PropertyMock (class in unittest.mock)
prot_c() (ftplib.FTP_TLS method)
prot_p() (ftplib.FTP_TLS method)
proto (socket.socket attribute)
protocol
  CGI
  FTP, [1]
  HTTP, [1], [2], [3], [4], [5]
  IMAP4
  IMAP4_SSL
  IMAP4_stream
  NNTP
  POP3
  SMTP
  Telnet

PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequence
PySequ
pthread_sigmask() (in module signal)

pthreads

pty

module

pty (module)

pu() (in module turtle)

publicId (xml.dom.DocumentType attribute)

PullDom (class in xml.dom.pulldom)

punctuation (in module string)

punctuation_chars (shlex.shlex attribute)

PurePath (class in pathlib)

PurePath.anchor (in module pathlib)

PurePath.drive (in module pathlib)

PurePath.name (in module pathlib)

PurePath.parent (in module pathlib)

PurePath.parents (in module pathlib)

PurePath.parts (in module pathlib)

PurePath.root (in module pathlib)

PurePath.stem (in module pathlib)

PurePath.suffix (in module pathlib)

PurePath.suffixes (in module pathlib)

PurePosixPath (class in pathlib)

PureProxy (class in smtpd)

PureWindowsPath (class in pathlib)

purge() (in module re)

Purpose.CLIENT_AUTH (in module ssl)

Purpose.SERVER_AUTH (in module ssl)

push() (asyncio.async_chat method)

    (code.InteractiveConsole method)

    (contextlib.ExitStack method)

push_alignment() (formatter.formatter method)

push_async_callback() (contextlib.AsyncExitStack method)

push_async_exit() (contextlib.AsyncExitStack method)
method)
push_font() (formatter.formatter method)
push_margin() (formatter.formatter method)
push_source() (shlex.shlex method)
push_style() (formatter.formatter method)
push_token() (shlex.shlex method)
push_with_producer() (asynchat.async_chat method)
pushbutton() (msilib.Dialog method)
put() (asyncio.Queue method)
    (multiprocessing.Queue method)
    (multiprocessing.SimpleQueue method)
    (queue.Queue method)
    (queue.SimpleQueue method)
put_nowait() (asyncio.Queue method)
    (multiprocessing.Queue method)
    (queue.Queue method)
    (queue.SimpleQueue method)
putch() (in module msvcrt)
putenv() (in module os)
putheader() (http.client.HTTPConnection method)
putp() (in module curses)
putrequest() (http.client.HTTPConnection method)
putwch() (in module msvcrt)
putwin() (curses.window method)
pvariance() (in module statistics)
pwd
    module
pwd (module)
pwd() (ftplib.FTP method)
pwrite() (in module os)
pwritev() (in module os)
Py_ABS (C macro)
Py_AddPendingCall (C function)
Py_AddPendingCall()
Py_AtExit (C function)
Py_BEGIN_ALLOW_THREADS (C macro)
Py_BLOCK_THREADS (C macro)
Py_buffer (C type)
Py_buffer.buf (C member)
Py_buffer.format (C member)
Py_buffer.internal (C member)
Py_buffer.itemsize (C member)
Py_buffer.len (C member)
Py_buffer.ndim (C member)
Py_buffer.obj (C member)
Py_buffer.readonly (C member)
Py_buffer.shape (C member)
Py_buffer.strides (C member)
Py_buffer.suboffsets (C member)
Py_BuildValue (C function)
Py_CallbackWarningFlag (C variable)
Py_CHARMASK (C macro)
Py_CLEAR (C function)
Py_compile (module)
 Py_COMPILED (in module imp)
Py_CompileString (C function)
Py_CompileString(), [1], [2]
Py_CompileStringExFlags (C function)
Py_CompileStringFlags (C function)
Py_CompileStringObject (C function)
Py_complex (C type)
Py_DebugFlag (C variable)
Py_DecodeLocale (C function)
Py_DECREF (C function)
Py_DECREF()
Py_DontWriteBytecodeFlag (C variable)
Py_Ellipsis (C variable)  PEI
Py_EncodeLocale (C function)  PEI
Py_END_ALLOW_THREADS (C macro)  [9], [17]
Py_EndInterpreter (C function)  PEI
Py_EnterRecursiveCall (C function)  PEI
Py_eval_input (C variable)  PEI
Py_Exit (C function)  PEI
Py_False (C variable)  PEI
Py_FatalError (C function)  PEI
Py_FatalError()  PEI
Py_FdIsInteractive (C function)  PEI
Py_file_input (C variable)  PEI
Py_Finalize (C function)  PEI
Py_FinalizeEx (C function)  PEI
Py_FinalizeEx(), [1], [2], [3], [4]  PEI
PY_FROZEN (in module imp)  [9]
Py_FrozenFlag (C variable)  PEI
Py_GetBuildInfo (C function)  PEI
Py_GetCompiler (C function)  PEI
Py_GetCopyright (C function)  PEI
Py_GETENV (C macro)  PEI
Py_GetExecPrefix (C function)  PEI
Py_GetExecPrefix()  PEI
Py_GetPath (C function)  PEI
Py_GetPath(), [1], [2]  PEI
Py_GetPlatform (C function)  PEI
Py_GetPrefix (C function)  PEI
Py_GetPrefix()  PEI
Py_GetProgramFullPath (C function)  PEI
Py_GetProgramFullPath()  PEI
Py_GetProgramName (C function)  PEI
Py_GetPythonHome (C function)  PEI
Py_GetVersion (C function)  PEI
Py_HashRandomizationFlag (C variable)
Py_IgnoreEnvironmentFlag (C variable)
Py_INCREF (C function)
Py_INCREF()
Py_Initialize (C function)
Py_Initialize(), [1], [2], [3]
Py_InitializeEx (C function)
Py_InspectFlag (C variable)
Py_InteractiveFlag (C variable)
Py_IsInitialized (C function)
Py_IsInitialized()
Py_IsolatedFlag (C variable)
Py_LeaveRecursiveCall (C function)
Py_LegacyWindowsFSEncodingFlag (C variable)
Py_LegacyWindowsStdioFlag (C variable)
Py_Main (C function)
Py_MAX (C macro)
Py_MEMBER_SIZE (C macro)
Py_MIN (C macro)
Py_mod_create (C variable)
Py_mod_exec (C variable)
Py_NewInterpreter (C function)
Py_None (C variable)
Py_NoSiteFlag (C variable)
Py_NotImplemented (C variable)
Py_NoUserSiteDirectory (C variable)
py_object (class in ctypes)
Py_OptimizeFlag (C variable)
Py_PRINT_RAW
Py_QuietFlag (C variable)
Py_REFCNT (C macro)
Py_ReprEnter (C function)
Py_ReprLeave (C function)
Py_RETURN_FALSE (C macro)
Py_RETURN_NONE (C macro)  
Py_RETURN_NOTIMPLEMENTED (C macro)  
Py_RETURN_RICHCOMPARE (C function)  
Py_RETURN_TRUE (C macro)  
Py_SetPath (C function)  
Py_SetPath()  
Py_SetProgramName (C function)  
Py_SetProgramName(), [1], [2], [3]  
Py_SetPythonHome (C function)  
Py_SetStandardStreamEncoding (C function)  
Py_single_input (C variable)  
Py_SIZE (C macro)  
PY_SOURCE (in module imp)  
PY_SSIZE_T_MAX  
Py_STRINGIFY (C macro)  
Py_TPFLAGS_BASE_EXC_SUBCLASS (built-in variable)  
Py_TPFLAGS_BASETYPE (built-in variable)  
Py_TPFLAGS_BYTES_SUBCLASS (built-in variable)  
Py_TPFLAGS_DEFAULT (built-in variable)  
Py_TPFLAGS_DICT_SUBCLASS (built-in variable)  
Py_TPFLAGS_HAVE_FINALIZE (built-in variable)  
Py_TPFLAGS_HAVE_GC (built-in variable)  
Py_TPFLAGS_HEAPTYPE (built-in variable)  
Py_TPFLAGS_LIST_SUBCLASS (built-in variable)  
Py_TPFLAGS_LONG_SUBCLASS (built-in variable)  
Py_TPFLAGS_READY (built-in variable)  
Py_TPFLAGS_READYING (built-in variable)  
Py_TPFLAGS_TUPLE_SUBCLASS (built-in variable)  
Py_TPFLAGS_TYPE_SUBCLASS (built-in variable)  
Py_TPFLAGS_UNICODE_SUBCLASS (built-in variable)  
PEP 436  
PEP 441  
PEP 442  
PEP 443  
PEP 445  
PEP 446  
PEP 448  
PEP 450  
PEP 451  
PEP 453  
PEP 454  
PEP 456  
PEP 461  
PEP 465  
PEP 466  
PEP 468  
PEP 471  
PEP 475  
PEP 476  
PEP 477  
PEP 478  
PEP 479  
PEP 483  
PEP 484  
PEP 485  
PEP 486  
PEP 487
Py_tracefunc (C type)
Py_True (C variable)
Py_tss_NEEDS_INIT (C macro)
Py_tss_t (C type)
Py_TYPE (C macro)
Py_UCS1 (C type)
Py_UCS2 (C type)
Py_UCS4 (C type)
Py_UNBLOCK_THREADS (C macro)
Py_UnbufferedStdioFlag (C variable)
Py_UNICODE (C type)
Py_UNICODE_IS_HIGH_SURROGATE (C macro)
Py_UNICODE_IS_LOW_SURROGATE (C macro)
Py_UNICODE_IS_SURROGATE (C macro)
Py_UNICODE_ISALNUM (C function)
Py_UNICODE_ISALPHA (C function)
Py_UNICODE_ISDECIMAL (C function)
Py_UNICODE_ISDIGIT (C function)
Py_UNICODE_ISLINEBREAK (C function)
Py_UNICODE_ISLOWER (C function)
Py_UNICODE_ISNUMERIC (C function)
Py_UNICODE_ISPRINTABLE (C function)
Py_UNICODE_ISSPACE (C function)
Py_UNICODE_ISTITLE (C function)
Py_UNICODE_ISUPPER (C function)
Py_UNICODE_JOIN_SURROGATES (C macro)
Py_UNICODE_TODECIMAL (C function)
Py_UNICODE_TODIGIT (C function)
Py_UNICODE_TOLOWER (C function)
Py_UNICODE_TOUNUMERIC (C function)
Py_UNICODE_TOTITLE (C function)
Py_UNICODE_TOUPPER (C function)
Py_UNREACHABLE (C macro)
Py_UNUSED (C macro)
Py_VaBuildValue (C function)
Py_VerboseFlag (C variable)
Py_VISIT (C function)
Py_XDECREF (C function)
Py_XDECREF()
Py_XDECREF (C function)
Py_XINCREF (C function)
PyAnySet_Check (C function)
PyAnySet_CheckExact (C function)
PyArg_Parse (C function)
PyArg_ParseTuple (C function)
PyArg_ParseTuple()
PyArg_ParseTupleAndKeywords (C function)
PyArg_ParseTupleAndKeywords()
PyArg_UnpackTuple (C function)
PyArg_ValidateKeywordArguments (C function)
PyArg_VaParse (C function)
PyArg_VaParseTupleAndKeywords (C function)
PyASCIIObject (C type)
PyAsyncMethods (C type)
PyAsyncMethods.am_aiter (C member)
PyAsyncMethods.am_anext (C member)
PyAsyncMethods.am_await (C member)
PyBool_Check (C function)
PyBool_FromLong (C function)
PyBUF_ANY_CONTIGUOUS (C macro)
PyBUF_C_CONTIGUOUS (C macro)
PyBUF_CONTIG (C macro)
PyBUF_CONTIG_RO (C macro)
PyBUF_F_CONTIGUOUS (C macro)
PyBUF_FORMAT (C macro)
PyBUF_FULL (C macro)
PyBUF_FULL_RO (C macro)
PyBUF_INDIRECT (C macro)
PyBUF_ND (C macro), [1]
Py_ASCIIObject (C type)
PyAsyncMethods (C type)
PyAsyncMethods.am_aiter (C member)
PyAsyncMethods.am_anext (C member)
PyAsyncMethods.am_await (C member)
PyBool_Check (C function)
PyBool_FromLong (C function)
PyBUF_ANY_CONTIGUOUS (C macro)
PyBUF_C_CONTIGUOUS (C macro)
PyBUF_CONTIG (C macro)
PyBUF_CONTIG_RO (C macro)
PyBUF_F_CONTIGUOUS (C macro)
PyBUF_FORMAT (C macro)
PyBUF_FULL (C macro)
PyBUF_FULL_RO (C macro)
PyBUF_INDIRECT (C macro)
PyBUF_ND (C macro)
Python Package Index (PyPI)
pypirc file
PYTHON*
python.function.entry (C function)
python.function.return (C function)
python_branch () (in module platform)
python_build() (in module platform)
python_compiler() (in module platform)
python_implementation() (in module platform)
python_is_optimized() (in module test.support)
python_revision() (in module platform)
python_version() (in module platform)
python_version_tuple() (in module platform)
PYTHONASYNCIODEBUG
PYTHONBREAKPOINT
PYTHONCASEOK
PYTHONCOERCECLOCALE
PYTHONDEBUG
PYTHONDONTWRITEBYTECODE
PYTHONDUMPREFS
PyBUF_RECORDS (C macro)
PyBUF_RECORDS_RO (C macro)
PyBUF_SIMPLE (C macro)
PyBUF_STRIDED (C macro)
PyBUF_STRIDED_RO (C macro)
PyBUF_STRIDES (C macro)
PyBUF_WRITABLE (C macro)
PyBuffer_FillContiguousStrides (C function)
PyBuffer_FillInfo (C function)
PyBuffer_IsContiguous (C function)
PyBuffer_Release (C function)
PyBuffer_SizeFromFormat (C function)
PyBuffer_ToContiguous (C function)
PyBufferProcs (C type)
PyBufferProcs.bf_getbuffer (C member)
PyBufferProcs.bf_releasebuffer (C member)
PyObject_ARRAY_AS_STRING (C function)
PyObject_AsString (C function)
PyObject_AsStringAndSize (C function)
PyObject_Check (C function)
PyObject_CheckExact (C function)
PyObject_Concat (C function)
PyObject_FromObject (C function)
PyObject_FromStringAndSize (C function)
PyObject_GET_SIZE (C function)
PyObject_RESIZE (C function)
PyObject_Size (C function)
PyObject_Type (C variable)
PyObjectObject (C type)
PyBytes_AS_STRING (C function)
PyBytes_AsString (C function)
PyBytes_AsStringAndSize (C function)
PyBytes_Check (C function)
PyBytes_CheckExact (C function)
PyThread_create_key (C function)
PyBytes_Concat (C function)
PyBytes_ConcatAndDel (C function)
PyBytes_FromFormat (C function)
PyBytes_FromFormatV (C function)
PyBytes_FromObject (C function)
PyBytes_FromString (C function)
PyBytes_FromStringAndSize (C function)
PyBytes_GET_SIZE (C function)
PyBytes_Size (C function)
PyBytes_Type (C variable)
PyBytesObject (C type)
PyCallable_Check (C function)
PyCallIter_Check (C function)
PyCallIter_New (C function)
PyCallIter_Type (C variable)
PyCapsule (C type)
PyCapsule_CheckExact (C function)
PyCapsule_Destructor (C type)
PyCapsule_GetContext (C function)
PyCapsule_GetDestructor (C function)
PyCapsule_GetName (C function)
PyCapsule_GetPointer (C function)
PyCapsule_Import (C function)
PyCapsule_IsValid (C function)
PyCapsule_New (C function)
PyCapsule_SetContext (C function)
PyCapsule_SetDestructor (C function)
PyCapsule_SetName (C function)
PyCapsule_SetPointer (C function)
PyCell_Check (C function)
PyCell_GET (C function)
PyCell_Get (C function)
PyCell_New (C function)
PyCell_SET (C function)
PyThread_delete_key (C function)
PyThread_delete_key_value (C function)
PyThread_get_key_value (C function)
PyThread_ReInitTLS (C function)
PyThread_set_key_value (C function)
PyThread_tss_alloc (C function)
PyThread_tss_create (C function)
PyThread_tss_delete (C function)
PyThread_tss_free (C function)
PyThread_tss_get (C function)
PyThread_tss_is_created (C function)
PyThread_tss_set (C function)
PyThreadState (C type)
PyThreadState_Clear (C function)
PyThreadState_Delete (C function)
PyThreadState_Get (C function)
PyThreadState_GetDict (C function)
PyThreadState_New (C function)
PyThreadState_Next (C function)
PyThreadState_SetAsyncExc (C function)
PyThreadState_Swap (C function)
PyTime_Check (C function)
PyTime_CheckExact (C function)
PyTime_FromTime (C function)
PyTimeZone_FromOffset (C function)
PyTimeZone_FromOffsetAndName (C function)
PyTrace_C_CALL (C variable)
PyTrace_C_EXCEPTION (C variable)
PyTrace_C_RETURN (C variable)
PyTrace_CALL (C variable)
PyTrace_EXCEPTION (C variable)
PyTrace_LINE (C variable)
PyCell_Set (C function)  
PyCell_Type (C variable)  
PyCellObject (C type)  
PyCFunction (C type)  
PyCFunctionWithKeywords (C type)  
PycInvalidationMode (class in py_compile)  
pyclbr (module)  
PyCode_Check (C function)  
PyCode_GetNumFree (C function)  
PyCode_New (C function)  
PyCode_NewEmpty (C function)  
PyCode_Type (C variable)  
PyCodec_BackslashReplaceErrors (C function)  
PyCodec_Decode (C function)  
PyCodec_Decoder (C function)  
PyCodec_Encode (C function)  
PyCodec_ENCODER (C function)  
PyCodec_IgnoreErrors (C function)  
PyCodec_IncrementalDecoder (C function)  
PyCodec_IncrementalEncoder (C function)  
PyCodec_KnownEncoding (C function)  
PyCodec_LookupError (C function)  
PyCodec_NameReplaceErrors (C function)  
PyCodec_Register (C function)  
PyCodec_RegisterError (C function)  
PyCodec_ReplaceErrors (C function)  
PyCodec_StreamReader (C function)  
PyCodec_StreamWriter (C function)  
PyCodec_StrictErrors (C function)  
PyCodec_XMLCharRefReplaceErrors (C function)  
PyCodeObject (C type)  
PyCompactUnicodeObject (C type)  
PyCompileError  
PyCompilerFlags (C type)
PyComplex_AsCComplex (C function)
PyComplex_Check (C function)
PyComplex_CheckExact (C function)
PyComplex_FromCComplex (C function)
PyComplex_FromDoubles (C function)
PyComplex_ImagAsDouble (C function)
PyComplex_RealAsDouble (C function)
PyComplex_Type (C variable)
PyComplexObject (C type)
PyContext (C type)
PyContext_CheckExact (C function)
PyContext_ClearFreeList (C function)
PyContext_Copy (C function)
PyContext_CopyCurrent (C function)
PyContext_Enter (C function)
PyContext_Exit (C function)
PyContext_New (C function)
PyContext_Type (C variable)
PyContextToken (C type)
PyContextToken_CheckExact (C function)
PyContextToken_Type (C variable)
PyContextVar (C type)
PyContextVar_CheckExact (C function)
PyContextVar_Get (C function)
PyContextVar_New (C function)
PyContextVar_Reset (C function)
PyContextVar_Set (C function)
PyContextVar_Type (C variable)
PyCoro_CheckExact (C function)
PyCoro_New (C function)
PyCoro_Type (C variable)
PyCoroObject (C type)
PyDate_Check (C function)
PyDate_CheckExact (C function)
PyDict_CheckExact (C function)
PyDict_Clear (C function)
PyDict_ClearFreeList (C function)
PyDict_Contains (C function)
PyDict_Copy (C function)
PyDict_DelItem (C function)
PyDict_DelItemString (C function)
PyDict_GetItem (C function)
PyDict_GetItemString (C function)
PyDict_GetItemWithError (C function)
PyDict_Items (C function)
PyDict_Keys (C function)
PyDict_Merge (C function)
PyDict_MergeFromSeq2 (C function)
PyDict_New (C function)
PyDict_Next (C function)
PyDict_SetDefault (C function)
PyDict_SetItem (C function)
PyDict_SetItemString (C function)
PyDict_Size (C function)
PyDict_Type (C variable)
PyDict_Update (C function)
PyDict_Values (C function)
PyDictObject (C type)
PyDictProxy_New (C function)
PyDLL (class in ctypes)
pydoc (module)
PyErr_BadArgument (C function)
PyErr_BadInternalCall (C function)
PyErr_CheckSignals (C function)
PyErr_Clear (C function)
PyErr_Clear(), [1]
PyErr_ExceptionMatches (C function)
PyErr_ExceptionMatches()
PyErr_Fetch (C function)
PyErr_Fetch()
PyErr_Format (C function)
PyErr_FormatV (C function)
PyErr_GetExcInfo (C function)
PyErr_GivenExceptionMatches (C function)
PyErr_NewException (C function)
PyErr_NewExceptionWithDoc (C function)
PyErr_NoMemory (C function)
PyErr_NormalizeException (C function)
PyErr_Occurred (C function)
PyErr_Occurred()
PyErr_Print (C function)
PyErr_PrintEx (C function)
PyErr_ResourceWarning (C function)
PyErr_Restore (C function)
PyErr_Restore()
PyErr_SetExcFromWindowsErr (C function)
PyErr_SetExcFromWindowsErrWithFilename (C function)
PyErr_SetExcFromWindowsErrWithFilenameObject (C function)
PyErr_SetExcFromWindowsErrWithFilenameObjects (C function)
PyErr_SetExcInfo (C function)
PyErr_SetFromErrno (C function)
PyErr_SetFromErrnoWithFilename (C function)
PyErr_SetFromErrnoWithFilenameObject (C function)
PyErr_SetFromErrnoWithFilenameObjects (C function)
PyErr_SetFromWindowsErr (C function)
PyErr_SetFromWindowsErrWithFilename (C function)
PyEval_ReleaseThread()
PyEval_RestoreThread (C function)
PyEval_RestoreThread(), [1]
PyEval_SaveThread (C function)
PyEval_SaveThread(), [1]
PyEval_SetProfile (C function)
PyEval_SetTrace (C function)
PyEval_ThreadsInitialized (C function)
PyExc_ArithmeticError
PyExc_AssertionError
PyExc_AttributeError
PyExc_BaseException
PyExc_BlockingIOError
PyExc_BrokenPipeError
PyExc_BufferError
PyExc_BytesWarning
PyExc_ChildProcessError
PyExc_ConnectionAbortedError
PyExc_ConnectionError
PyExc_ArithmeticError
PyExc_AssertionError
PyExc_AttributeError
PyExc_BaseException
PyExc_BlockingIOError
PyExc_BrokenPipeError
PyExc_BufferError
PyExc_BytesWarning
PyExc_ChildProcessError
PyExc_ConnectionAbortedError
PyExc_ConnectionError
PyUnicc function
PyUnicc function
PyUnicc function
PyUnicc function
PyUnicc function
PyUnicc function
PyUnicc function
PyVarOl
PyVarOl
PyVarOl
PyVarOl
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWeak
PyWrap
PyZipFil
qiflush() (in module curses)
QName (class in xml.etree.ElementTree)
qsize() (asyncio.Queue method)
          (multiprocessing.Queue method)
          (queue.Queue method)
          (queue.SimpleQueue method)
**qualified name**
quantize() (decimal.Context method)
          (decimal.Decimal method)
QueryInfoKey() (in module winreg)
QueryReflectionKey() (in module winreg)
QueryValue() (in module winreg)
QueryValueEx() (in module winreg)
Queue (class in asyncio)
          (class in multiprocessing)
          (class in queue)
queue (module)
          (sched.scheduler attribute)
Queue()
          (multiprocessing.managers.SyncManager method)
QueueEmpty
QueueFull
QueueHandler (class in logging.handlers)
QueueListener (class in logging.handlers)  
quick_ratio()
          (difflib.SequenceMatcher method)
quit (built-in variable)
          (pdb command)
quit()  (ftplib.FTP method)
          (nntplib.NNTP method)
          (poplib.Popup method)
          (smtplib.SMTP method)
quopri (module)
quote() (in m email.utils)
          (in module shlex)
          (in module urllib.parse)
QUOTE_ALL (in module csv)
quote_from_bytes() (module urllib.parse)
QUOTE_MINIMAL (module csv)
QUOTE_NONE (module csv)
QUOTE_NONNUM (in module csv)
quote_plus() (in m urllib.parse)
quoteattr() (in m
xml.sax.saxutils
quotechar  (csv.D
attribute)
quotted-printable
    encoding
quotes      (shlex.
attribute)
quoting     (csv.D
attribute)
R

r"
raw string literal
r'
raw string literal
R_OK (in module os)
radians() (in module math)
(in module turtle)
RadioButtonGroup (class in msilib)
radiogroup() (msilib.Dialog method)
radx() (decimal.Context method)
(decimal.Decimal method)
RADIXCHAR (in module locale)
raise
    statement, [1]
raise (2to3 fixer)
raise an exception
raise_on_defect (email.policy.Policy attribute)
RAISE_VARARGS (opcode)
raising
    exception
RAND_add() (in module ssl)
RAND_bytes() (in module ssl)
RAND_egd() (in module ssl)
RAND_pseudo_bytes() (in module ssl)
RAND_status() (in module ssl)
randbelow() (in module secrets)
randbits() (in module secrets)
randint() (in module random)
random (module)
random() (in module random)
randrange() (in module random)
r
requires_lzma()
requires_mac_test.support)
requires_resou
requires_zlib()
reserved (zipfil
reserved word
RESERVED_F
RESERVED_\n
reset() (bdb.Bc
reset_mock() (c
reset_prog_mode()
reset_shell_mode()
resetbuffer() (c
resetlocale() (i
built-in function
object
range (built-in class)
RARROW (in module token)
ratecv() (in module audioop)
ratio() (difflib.SequenceMatcher method)
Rational (class in numbers)
raw (io.BufferedIOBase attribute)
raw string
raw() (in module curses)
raw_data_manager (in module email.contentmanager)
raw_decode() (json.JSONDecoder method)
raw_input (2to3 fixer)
raw_input() (code.InteractiveConsole method)
RawArray() (in module multiprocessing.sharedctypes)
RawConfigParser (class in configparser)
RawDescriptionHelpFormatter (class in argparse)
RawIOBase (class in io)
RawPen (class in turtle)
RawTextHelpFormatter (class in argparse)
RawTurtle (class in turtle)
RawValue () (in module multiprocessing.sharedctypes)
RBRACE (in module token)
rcpttos (smtpd.SMTPChannel attribute)
re (module)
    (re.Match attribute)
read () (asyncio.StreamReader method)
    (chunk.Chunk method)
resetscreen () (in module curses)
resetty() (in module curses)
resetwarnings() (in module curses.mmap.mmap)
resize() (curses.resize () (in module curses.mmap.mmap)
resize_term() (in module curses)
resizemode() (in module curses)
resolution() (datetime.resolution () (datetime.datetime.resolution())
resolutionterm() (in module curses)
resolution (datetime)
resolve() (pathlib.Path resolve () (pathlib.Path)
resolve_bases()
resolve_name()
resolveEntity()
method)
Resource () (importlib.abc.ResourceReader
ResourceDenied
ResourceLoader (class in importlib.abc)
ResourceReader (class in importlib.abc)
ResourceWarning
response () (nntplib.NNTPError
response () (imaplib.IMAP4
ResponseNotReady
responses
(http.server.BaseHTTPRequestHandler
restart (pdb command)
response (http.server.response)
attribute)
(codecs.StreamReader method) restore() (in module readline)
(configparser.ConfigParser method) RESTRICTED
(http.client.HTTPResponse method) restricted
(imaplib.IMAP4 method) execution
(in module os) restype (ctypes._FuncPtr attribute)
(io.BufferedIOBase method) result() (asyncio.Future method)
(io.BufferedReader method) (asyncio.Task method)
(io.RawIOBase method) results() (concurrent.futures.Future method)
(io.TextIOBase method) resume_reading()
(mimetypes.MimeTypes method) resume_writin⋯
(mmap.mmap method) retre()
(ossaudiodev.oss_audio_device method) retrbinary() (ftp⋯
(ssl.MemoryBIO method) retrlines() (ftplib.FTP method)
(ssl.SSLSocket method) return⋯
(urllib.robotparser.RobotFileParser method) return
read() (io.BufferedIOBase method) return (pdb.cor⋯
(read_all() (telnetlib.Telnet method) return_annotat⋯
(read_binary() (in module importlib.resources) return_ok()⋯
(read_byte() (mmap.mmap method) method)
(read_bytes() (pathlib.Path method) RETURN_VAL
(read_dict() (configparser.ConfigParser⋯ return_value (⋯
(read_eager() (telnetlib.Telnet method) returncode (asyncio.async⋯
(read_environ() (in module wsgiref.handlers) subproce⋯
(read_events() (subproce⋯
(xml.etree.ElementTree.XMLPullParser reverse() (array.⋯
(method) collection
(read_file() (configparser.ConfigParser⋯ (in module readli⋯
(read_history_file() (in module readline) (sequence⋯
read_init_file() (in module readline)
read_lazy() (telnetlib.Telnet method)
read_mime_types() (in module mimetypes)
READ_RESTRICTED
read_sb_data() (telnetlib.Telnet method)
read_some() (telnetlib.Telnet method)
read_string() (configparser.ConfigParser method)
read_text() (in module importlib.resources)
(readlib.Path method)
read_token() (shlex.shlex method)
read_until() (telnetlib.Telnet method)
read_very_eager() (telnetlib.Telnet method)
read_very_lazy() (telnetlib.Telnet method)
read_windows_registry()
(mimetypes.MimeTypes method)
READABLE (in module tkinter)
(readable() (asyncore.dispatcher method)
(pathlib.Path method)
read_token() (shlex.shlex method)
read_until() (telnetlib.Telnet method)
read_very_eager() (telnetlib.Telnet method)
read_very_lazy() (telnetlib.Telnet method)
read_windows_registry()
(mimetypes.MimeTypes method)
READABLE (in module tkinter)
(readable() (asyncore.dispatcher method)
(pathlib.Path method)
read_all() (io.RawIOBase method)
read() (in module csv)
ReadError
readexactly() (asyncio.StreamReader method)
readfp() (configparser.ConfigParser method)
(readlib.Path method)
readframes() (aifc.aifc method)
(sunau.AU_read method)
(wave.Wave_read method)
readinto() (http.client.HTTPResponse method)
(http.client.HTTPResponse method)
(ioutil.BufferedIOBase method)
(ioutil.BytesIO method)
readline (module)
readline() (asyncio.StreamReader method) RFC 2060
   (codecs.StreamReader method) [9], [10], [1]
   (distutils.text_file.TextFile method) RFC 2068
   (imaplib.IMAP4 method) RFC 2104
   (io.IOBase method) RFC 2109
   (io.TextIOBase method) [9], [10], [1]
   (mmap.mmap method) RFC 2183
readlines() (codecs.StreamReader method) RFC 2231
   (distutils.text_file.TextFile method) [9], [10], [1]
   (io.IOBase method) RFC 2295
readlink() (in module os) RFC 2342
readmodule() (in module pycbr) RFC 2368
readmodule_ex() (in module pycbr) RFC 2373
READONLY RFC 2396
readonly (memoryview attribute) RFC 2397
readPlist() (in module plistlib) RFC 2449
readPlistFromBytes() (in module plistlib) RFC 2487
ReadTransport (class in asyncio) RFC 2518
readuntil() (asyncio.StreamReader method) RFC 2595
readv() (in module os) RFC 2616
ready() (multiprocessing.poolAsyncResult method) [9]
Real (class in numbers) RFC 2774
real (numbers.Complex attribute) RFC 2818
Real Media File Format RFC 2821
real_max_memuse (in module test.support) RFC 2822
real_quick_ratio() (difflib.SequenceMatcher method) [9], [10], [1]
realloc() RFC 2964
realpath() (in module os.path) [9], [10], [17], [18], [19]
REALTIME_PRIORITY_CLASS (in module subprocess) RFC 2980
reap_children() (in module test.support) RFC 3056
reap_threads() (in module test.support) RFC 3171
reason (http.client.HTTPResponse attribute) RFC 3207
  (UnicodeError attribute) RFC 3229
  (ssl.SSLError attribute) RFC 3280
  (urllib.error.HTTPError attribute) RFC 3330
  (urllib.error.URLError attribute) RFC 3339
reattach() (tkinter.ttk.Treeview method) RFC 3454
rebinding
  name RFC 3490
reccontrols() (ossaudiodev.oss_mixer_device method) RFC 3492
received_data (smtplib.SMTPChannel attribute) RFC 3501
received_lines (smtplib.SMTPChannel attribute) RFC 3548
recent() (imaplib.IMAP4 method) RFC 3549
reconfigure() (io.TextIOWrapper method) RFC 3542
record_original_stdout() (in module test.support) RFC 3977
records (unittest.TestCase attribute) RFC 4086
rect() (in module cmath) RFC 4122
rectangle() (in module curses.textpad) RFC 4180
RecursionError RFC 4193
recursive_repr() (in module reprlib) RFC 4217
recv() (asyncore.dispatcher method) RFC 4291
  (multiprocessing.connection.Connection method) RFC 4380
  (socket.socket method) RFC 4627
recv_bytes() RFC 4918
  (multiprocessing.connection.Connection method) RFC 4954
recv_bytes_into() RFC 5161
  (multiprocessing.connection.Connection method) RFC 5246
recv_into() (socket.socket method) RFC 5280
recv_into() (socket.socket method) RFC 5321
recvfrom() (socket.socket method) RFC 5322
recvfrom_into() (socket.socket method) [9], [10],
recvmsg() (socket.socket method) [17], [18],
recvmsg_into() (socket.socket method) RFC 5424
redirect_request() RFC 5424
(uurlib.request.HTTPRedirectHandler method) RFC 5735
redirect_stderr() (in module contextlib) RFC 5842
redirect_stdout() (in module contextlib) RFC 5929
redisplay() (in module readline) RFC 6066
redrawln() (curses.window method) RFC 6125
redrawwin() (curses.window method) RFC 6152
reduce (2to3 fixer) RFC 6531
reduce() (in module functools) RFC 6532
ref (class in weakref) RFC 6585
refcount_test() (in module test.support) RFC 6855
reference RFC 6856
    attribute RFC 7159
reference count RFC 7230
reference counting RFC 7231
ReferenceError, [1] [9], [10],
ReferenceTypeError (in module weakref) [17], [18],
refold_source (email.policy.EmailPolicy [25], [26],
attribute) [33], [34]
refresh() (curses.window method) RFC 7232
REG_BINARY (in module winreg) RFC 7233
REG_DWORD (in module winreg) RFC 7235
REG_DWORD_BIG_ENDIAN (in module winreg) RFC 7238
REG_DWORD_LITTLE_ENDIAN (in module winreg) RFC 7301
REG_EXPAND_SZ (in module winreg) RFC 7525
REG_FULL_RESOURCE_DESCRIPTOR (in module winreg) RFC 7914
REG_LINK (in module winreg) RFC 821,
REG_MULTI_SZ (in module winreg)
REG_NONE (in module winreg)
REG_QWORD (in module winreg)
REG_QWORD_LITTLE_ENDIAN (in module winreg)
REG_RESOURCE_LIST (in module winreg)
REG_RESOURCE_REQUIREMENTS_LIST (in module winreg)
REG_SZ (in module winreg)
register() (abc.ABCMeta method)
    (in module atexit)
    (in module codecs)
    (in module faulthandler)
    (in module webbrowser)
    (multiprocessing.managers.BaseManager method)
    (select.devpoll method)
    (select.epoll method)
    (select.poll method)
    (selectors.BaseSelector method)
register_adapter() (in module sqlite3)
register_archive_format() (in module shutil)
register_at_fork() (in module os)
register_converter() (in module sqlite3)
register_defect() (email.policy.Policy method)
register_dialect() (in module csv)
register_error() (in module codecs)
register_function() (in module codecs)
    (xmlrpc.server.CGIXMLRPCRequestHandler method)
    (xmlrpc.server.SimpleXMLRPCServer method)
register_instance() (xmlrpc.server.CGIXMLRPCRequestHandler method)
rfi
method)
       (xmlrpc.server.SimpleXMLRPCServer method)
register_introspection_functions()
(xmlrpc.server.CGIXMLRPCRequestHandler method)
       (xmlrpc.server.SimpleXMLRPCServer method)
register_multicall_functions()
(xmlrpc.server.CGIXMLRPCRequestHandler method)
       (xmlrpc.server.SimpleXMLRPCServer method)
register_namespace()  (in  module xml.etree.ElementTree)
register_optionflag()  (in  module doctest)
register_shape()  (in  module turtle)
register_unpack_format()  (in  module shutil)
registerDOMImplementation()  (in  module xml.dom)
registerResult()  (in  module unittest)
regular
       package
regular package
relative
       URL
       import
relative_to()  (pathlib.PurePath method)
release()  (_thread.lock method)
       (asyncio.Condition method)
       (asyncio.Lock method)
       (asyncio.Semaphore method)
       (in  module platform)
       (logging.Handler method)
rledecode_hqx
       RLIM_INFINITY
       RLIMIT_AS  (in
       RLIMIT_CORE
       RLIMIT_CPU  (RLIMIT_DATA
       RLIMITFSIZE
       RLIMIT_MEML
       RLIMIT_MSGMC
       RLIMIT_NICE
       RLIMIT_NOFIL
       RLIMIT_NPROC
       RLIMIT_NPTS
       RLIMIT_OFILE
       RLIMIT_RSS  (RLIMIT_RTPR
       RLIMIT_RT\n
       RLIMIT_SBSIZ
       RLIMIT_SIGP\f
       RLIMIT_STAC
       RLIMIT_SWAP

       RLIMN
       RLock  (class  ir
       (class  in  t\n
       RLock()
       (multiprocessir method)
rmd()  (ftplib.FT
rmdir()  (in  mod
gdir()  (in  mod
       (pathlib.P\c
RMFF
rms()  (in  modu
rmtree()  (in  mo
(memoryview method)
(multiprocessing.Lock method)
(multiprocessing.RLock method)
(threading.Condition method)
(threading.Lock method)
(threading.RLock method)
(threading.Semaphore method)
release_lock() (in module imp)
reload (2to3 fixer)
reload() (in module imp)
(in module importlib)
relpath() (in module os.path)
remainder() (decimal.Context method)
(in module math)
remainder_near() (decimal.Context method)
(decimal.Decimal method)
RemoteDisconnected
remove() (array.array method)
(collections.deque method)
(frozenset method)
(in module os)
(mailbox.MH method)
(mailbox.Mailbox method)
(sequence method)
(xml.etree.ElementTree.Element method)
remove_child_handler() (asyncio.AbstractChildWatcher method)
remove_done_callback() (asyncio.Future method)
(asynclio.Task method)
remove_flag() (mailbox.MaildirMessage method)
(mailbox.MMDFMessage method)
(mailbox.mboxMessage method)
(RobotFileParser class in urllib.robotparser)
rollback() (sqlite3.Connection method)
ROT_THREE (opcode)
 ROT_TWO (opcode)
rotate() (collections.deque method)
(decimal.Context method)
(decimal.Decimal method)
RotatingFileHandler (class in logging.handlers)
rotation_filename (logging.handlers.BaseRotatingHandler method)
rotator (logging.handlers.BaseRotatingHandler attribute)
round built-in function
round() (built-in function)
ROUND_05UP (in module decimal)
ROUND_CEILING (in module decimal)
ROUND_DOWN (in module decimal)
ROUND_FLOOR (in module decimal)
ROUND_HALF_DOWN (in module decimal)
ROUND_HALF_EVEN (in module decimal)
ROUND_HALF_UP (in module decimal)
ROUND_UP (in module decimal)
Rounded (class in decimal)
Row (class in sqlite3)
row_factory (sqlite3.Connection attribute)
rowcount (sqlite3.Cursor attribute)
RPAR (in module token)
rpartition() (bytearray method)
remove_folder() (mailbox.Maildir method)
  (mailbox.MH method)
remove_header()     (urllib.request.Request method)
remove_history_item() (in module readline)
remove_label()     (mailbox.BabylMessage method)
remove_option()   (configparser.ConfigParser method)
                  (optparse.OptionParser method)
remove_pyc()      (msilib.Directory method)
remove_reader()     (asyncio.loop method)
remove_section() (configparser.ConfigParser method)
remove_sequence() (mailbox.MHMessage method)
remove_signal_handler() (asyncio.loop method)
remove_tree()     (in module distutils.dir_util)
remove_writer() (asyncio.loop method)
removeAttribute() (xml.dom.Element method)
removeAttributeNode() (xml.dom.Element method)
removeAttributeNS() (xml.dom.Element method)
removeChild() (xml.dom.Node method)
removedirs()  (in module os)
removeFilter() (logging.Handler method)
                  (logging.Logger method)
removeHandler() (in module unittest)
                  (logging.Logger method)
removeResult()  (in module unittest)
removexattr() (in module os)
rename() (ftp.lib.FTP method)
ReplacePackage() (in module modulefinder)
report() (filecmp.dircmp method)
    (modulefinder.ModuleFinder method)
REPORT_CDIFF (in module doctest)
report_failure() (doctest.DocTestRunner method)
report_full_closure() (filecmp.dircmp method)
REPORT_NDIFF (in module doctest)
REPORT_ONLY_FIRST_FAILURE (in module doctest)
report_partial_closure() (filecmp.dircmp method)
report_start() (doctest.DocTestRunner method)
report_success() (doctest.DocTestRunner method)
REPORT_UDIFF (in module doctest)
report_unexpected_exception() (doctest.DocTestRunner method)
REPORTING_FLAGS (in module doctest)
repr
    built-in function, [1], [2], [3]
repr (2to3 fixer)
Repr (class in reprlib)
repr() (built-in function)
    __repr__() (object method)
repr() (in module reprlib)
    (reprlib.Repr method)
repr1() (reprlib.Repr method)
representation
    integer
reprlib (module)
Request (class in urllib.request)
request() (http.client.HTTPConnection...
method
request_queue_size
(socketserver.BaseServer attribute)
request_rate()
(urllib.robotparser.RobotFileParser method)
request_uri() (in module wsgiref.util)
request_version
(http.server.BaseHTTPRequestHandler attribute)
RequestHandlerClass
(socketserver.BaseServer attribute)
requestline
(http.server.BaseHTTPRequestHandler attribute)
requires() (in module test.support)
requires_bz2() (in module test.support)
requires_docstrings() (in module test.support)
requires_freebsd_version() (in module test.support)
requires_gzip() (in module test.support)
requires_IEEE_754() (in module test.support)
requires_linux_version() (in module test.support)
S (in module re)
S_ENFMT (in module stat)
S_IEXEC (in module stat)
S_IFBLK (in module stat)
S_IFCHR (in module stat)
S_IFDIR (in module stat)
S_IFDIR (in module stat)
S_IFDOOR (in module stat)
S_IFIFO (in module stat)
S_IFLNK (in module stat)
S_IFMT() (in module stat)
S_IFPORT (in module stat)
S_IFREG (in module stat)
S_IFSOCK (in module stat)
S_IFWHT (in module stat)
S_IMODE() (in module stat)
S_IREAD (in module stat)
S_IRGRP (in module stat)
S_IROTH (in module stat)
S_IRUSR (in module stat)
S_IRWXG (in module stat)
S_IRWXO (in module stat)
S_IRWXU (in module stat)
S_ISBLK() (in module stat)
S_ISCHR() (in module stat)
S_ISDIR() (in module stat)
S_ISDOOR() (in module stat)
S_ISFIFO() (in module stat)
S_ISGID (in module stat)
S_ISLNK() (in module stat)
S_ISPORT() (in module stat)
S_ISREG() (in module stat)
S_ISSOCK() (in module stat)

size() (ftplib.
size_diff (tracemalloc.
SIZE_MAX
Sized (class
test.support)
skip_unless_
test.support)
skip_unless_
skipUnless()
SLASH (in module
SLASHEQU.
sleep() (nn
(sleep() (in module
slice, [1]
assign
built-in f
object
S_ISUID (in module stat)  operatic
S_ISVTX (in module stat)  slice (built-in
S_ISWHT() (in module stat)  slicing, [1], [2]
S_IWGRP (in module stat)  assignment
S_IWRITE (in module stat)  SMTP
S_IWOTH (in module stat)  protocol
S_IWRITE (in module stat)  SMTP (class
S_IWUSR (in module stat)  (in module
S_IXGRP (in module stat)  smtp_server
S_IXOTH (in module stat)  SMTP_SSL
S_IXUSR (in module stat)  safe (uuid.SafeUUID attribute)
safe_substitute() (string.Template method)  smtp_state ()
SafeChildWatcher (class in asyncio)  SMTPAuthenti
saferepr() (in module pprint)  SMTPChan
SafeUUID (class in uuid)  SMTPConne
same_files (filecmp.dircmp attribute)  smtpd (modu
same_quantum() (decimal.Context method)  SMTPDataE
  (decimal.Decimal method)  SMTPHandl
samefile() (in module os.path)  SMTPHEloE
  (pathlib.Path method)  smtpplib (modu
SameFileError  SMTPNotSu
sameopenfile() (in module os.path)  SMTPRecipi
samestat() (in module os.path)  SMTPRespc
sample() (in module random)  SMTPSende
save() (http.cookiejar.FileCookieJar method)  SMTPServer
SAVEDCWD (in module test.support)  SMTPServer
SaveKey() (in module winreg)  SMTPUTF8
SaveSignals (class in test.support)  Snapshot (cl
savetty() (in module curses)  SND_ALIAS
SAX2DOM (class in xml.dom.pulldom)  SNDASYNC
SAXException  SND_Filen
SAXNotRecognizedException  SND_LOOP
SAXNotSupportedException  SND_MEMEC
SAXParseException  SND_NODE
scaleb() (decimal.Context method)
    (decimal.Decimal method)
scandir() (in module os)
scanf()
sched (module)
SCHED_BATCH (in module os)
SCHED_FIFO (in module os)
sched_get_priority_max() (in module os)
sched_get_priority_min() (in module os)
sched_getaffinity() (in module os)
sched_getparam() (in module os)
sched_getscheduler() (in module os)
SCHED_IDE (in module os)
SCHED_OTHER (in module os)
sched_param (class in os)
sched_priority (os.sched_param attribute)
SCHED_RESET_ON_FORK (in module os)
SCHED_RR (in module os)
sched_rr_get_interval() (in module os)
sched_setaffinity() (in module os)
sched_setparam() (in module os)
sched_setscheduler() (in module os)
SCHED_SPORADIC (in module os)
sched_yield() (in module os)
scheduler (class in sched)
schema (in module msilib)
scope, [1]
Screen (class in turtle)
screensize() (in module turtle)
script_from_examples() (in module doctest)
scroll() (curses.window method)
ScrolledCanvas (class in turtle)
scrollok() (curses.window method)
scrypt() (in module hashlib)

SND_NOSTOP (in module winsound)
SND_NOWAIT (in module winsound)
SND_PURGE (in module winsound)
sndhdr (module)
sni_callback
Sniffer (class)
sock_accept
SOCK_CLO
SOCK_DGR
SOCK_MAX
SOCK_NON
SOCK_RAW
SOCK_RDV
sock_recv()
sock_recv_from()
sock_sendfil
SOCK_SEQ
SOCK_STR
socket
socket (module)
socket (object)
socketpair()
sdterr
  stdin stdout
seal() (in module unittest.mock)
search
  path, module, [1], [2], [3], [4], [5], [6], [7]
search() (imaplib.IMAP4 method)
  (in module re)
  (re.Pattern method)
second (datetime.datetime attribute)
  (datetime.time attribute)
seconds since the epoch
secrets (module)
SECTCRE (configparser.ConfigParser attribute)
sections() (configparser.ConfigParser method)
secure (http.cookiejar.Cookie attribute)
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512
Secure Sockets Layer
security
  CGI
see() (tkinter.ttk.Treeview method)
seed() (in module random)
seek() (chunk.Chunk method)
  (io.IOBase method)
  (io.TextIOBase method)
  (mmap.mmap method)
SEEK_CUR (in module os)
SEEK_END (in module os)
SEEK_SET (in module os)
seekable() (io.IOBase method)
seen_greeting (smtpd.SMTPChannel attribute)
Select (class in tkinter.tix)
select (module)
select() (imaplib.IMAP4 method)
method)
  (socket.socket method)
send_bytes()
(multiprocessing.connection.Connection method)
send_error()
(http.server.BaseHTTPRequestHandler method)
send_flowing_data() (formatter.writer method)
send_header()
(http.server.BaseHTTPRequestHandler method)
send_hor_rule() (formatter.writer method)
send_label_data() (formatter.writer method)
send_line_break() (formatter.writer method)
send_literal_data() (formatter.writer method)
send_message() (smtplib.SMTP method)
send_paragraph() (formatter.writer method)
send_response()
(http.server.BaseHTTPRequestHandler method)
send_response_only()
(http.server.BaseHTTPRequestHandler method)
send_signal()
(asyncio.asyncio.subprocess.Process method)
  (asyncio.SubprocessTransport method)
  (subprocess.Popen method)
sendall() (socket.socket method)
sendcmd() (ftplib.FTP method)
sendfile() (asyncio.loop method)
  (in module os)
  (socket.socket method)
  (wsgiref.handlers.BaseHandler method)
SendfileNotAvailableError
sendmail() (smtplib.SMTP method)
sendmsg() (socket.socket method)
sendmsg_afalg() (socket.socket method)
sendto() (asyncio.DatagramTransport method)  
(socket.socket method)
sentinel (in module unittest.mock)  
(multiprocessing.Process attribute)
sep (in module os)
sequence
  item
  iteration
  object, [1], [2], [3], [4], [5], [6], [7], [8]
types, immutable
  types, mutable
  types, operations on, [1]
Sequence (class in collections.abc)  
(class in typing)
sequence (in module msilib)
sequence2st() (in module parser)
SequenceMatcher (class in difflib), [1]
serializing
  objects
serve_forever() (asyncio.Server method)  
(socketserver.BaseServer method)
sentinel (in module unittest.mock)  
(multiprocessing.Process attribute)
sep (in module os)
sequence
  item
  iteration
  object, [1], [2], [3], [4], [5], [6], [7], [8]
types, immutable
  types, mutable
  types, operations on, [1]
Sequence (class in collections.abc)  
(class in typing)
sequence (in module msilib)
sequence2st() (in module parser)
SequenceMatcher (class in difflib), [1]
serializing
  objects
serve_forever() (asyncio.Server method)  
(socketserver.BaseServer method)
server
  WWW, [1]
Server (class in asyncio)
  (http.server.BaseHTTPRequestHandler attribute)
server_activate()  
(socketserver.BaseServer method)
server_address  
(socketserver.BaseServer attribute)
server_bind()  
(socketserver.BaseServer method)
server_close()  
(socketserver.BaseServer method)
server_hostname (ssl.SSLSocket attribute)
server_side (ssl.SSLSocket attribute)
server_software (wsgiref.handlers.BaseHandler attribute)
server_version (http.server.BaseHTTPRequestHandler attribute)

ServerProxy (class in xmlrpc.client)
service_actions() (socketserver.BaseServer method)

session (ssl.SSLSocket attribute)
session_reused (ssl.SSLSocket attribute)
session_stats() (ssl.SSLContext method)

set display
object, [1], [2], [3]

set (built-in class)
Set (class in collections.abc)
(class in typing)

Set Breakpoint

set type object

set() (asyncio.Event method)
(configparser.ConfigParser method)
(configparser.RawConfigParser method)
(contextvars.ContextVar method)
(http.cookies.Morsel method)
(ossaudiodev.oss_mixer_device method)
(test.support.EnvironmentVarGuard method)
(threading.Event method)
(ttkinter.ttk.Combo box method)

st_size (os.stat_result attribute)
st_type (os.stat_result attribute)
ST_UID (in module stat)
st_uid (os.stat_result attribute)

stack
evaluation trace

stack()
(stack view)

stack_effect()

stack_size()
(stack)

stackable

streams

StackSummary

stamp() (in module turtle)

standard output

Standard C

standard input

standard_b64decode() (in module base64)

standard_b64encode() (in module base64)

standarderror (2to3 fixer)

standend() (in module curses)

standout() (in module curses)

STAR (in module token)

STAREQUAL (in module token)

starmap() (in module itertools)

(multiprocessing.pool.Pool method)

starmap_async() (multiprocessing.pool.Pool method)

start (range attribute)
set_continue() (bdb.Bdb method)
set_cookie() (http.cookiejar.CookieJar method)
set_cookie_if_ok() (http.cookiejar.CookieJar method)
set_coroutine_origin_tracking_depth() (in module sys)
set_coroutine_wrapper() (in module sys)
set_current() (msilib.Feature method)
set_data() (importlib.abc.SourceLoader method)
   (importlib.machinery.SourceFileLoader method)
set_date() (mailbox.MaildirMessage method)
set_debug() (asyncio.loop method)
   (in module gc)
set_debuglevel() (ftplib.FTP method)
   (http.client.HTTPConnection method)
   (ftplib.FTP method)
   (ntplib.NNTP method)
   (poplib.POP3 method)
   (smtplib.SMTP method)
   (telnetlib.Telnet method)
set_default_executor() (asyncio.loop method)
set_default_type()
(email.message.EmailMessage method)
   (email.message.Message method)
set_default_verify_paths() (ssl.SSLContext method)
set_defaults() (argparse.ArgumentParser method)
   (optparse.OptionParser method)
set_ecdh_curve() (ssl.SSLContext method)
set_errno() (in module ctypes)
set_event_loop()
   (asyncio.AbstractEventLoopPolicy method)
break, [:

set_event_loop_policy() (in module asyncio)

set_exception() (asyncio.Future method)
(concurrent.futures.Future method)

set_exception_handler() (asyncio.loop method)

set_executable() (in module multiprocessing)

set_executables() (distutils.ccompiler.CCompiler method)

set_flags() (mailbox.MaildirMessage method)
(mailbox.MMDFMessage method)
(mailbox.mboxMessage method)

set_from() (mailbox.mboxMessage method)
(mailbox.MMDFMessage method)

set_handle_inheritable() (in module os)

set_history_length() (in module readline)

set_include_dirs() (distutils.ccompiler.CCompiler method)

set_info() (mailbox.MaildirMessage method)

set_inheritable() (in module os)
(socket.socket method)

set_labels() (mailbox.BabylMessage method)

set_last_error() (in module ctypes)

set_libraries() (distutils.ccompiler.CCompiler method)

set_library_dirs() (distutils.ccompiler.CCompiler method)

set_link_objects() (distutils.ccompiler.CCompiler method)

set_literal (2to3 fixer)

set_loader() (in module importlib.util)

set_match_tests() (in module test.support)

set_memlimit() (in module test.support)

set_next() (bdb.Bdb method)

set_nonstandard_attr() (http.cookiejar.Cookie

break, [:

class

compou

continue

def

del, [1],

except

express

for, [1], |

future

global, [ |

if, [1]

import, |

loop, [1]

nonloca

pass

raise, [1

return, [ |

simple

try, [1], |

while, [1

with, [1

yield

statement gr

staticmethoc

built-in f

staticmethoc

Statistic (cl

StatisticDiff ( |

statistics (m |

statistics() (ti |

StatisticsErro

Stats (class |
method)
set_npn_protocols() (ssl.SSLContext method)
set_ok() (http.cookiejar.CookiePolicy method)
set_option_negotiation_callback()
       (telnetlib.Telnet method)
set_output_charset()  (gettext.NullTranslations
       method)
set_package() (in module importlib.util)
set_param()    (email.message.EmailMessage
       method)
               (email.message.Message method)
set_pasv() (ftplib.FTP method)
set_payload()    (email.message.Message
       method)
set_policy() (http.cookiejar.CookieJar method)
set_position() (xdrlib.Unpacker method)
set_pre_input_hook() (in module readline)
set_progress_handler()  (sqlite3.Connection
       method)
set_protocol() (asyncio.BaseTransport method)
set_proxy() (urllib.request.Request method)
set_python_build()    (in module
       distutils.sysconfig)
set_quit() (bdb.Bdb method)
set_recsrc() (ossaudiodev.oss_mixer_device
       method)
set_result() (asyncio.Future method)
                (concurrent.futures.Future method)
set_return() (bdb.Bdb method)
set_running_or_notify_cancel()  
       (concurrent.futures.Future method)
set_runtime_library_dirs()  
       (distutils.ccompiler.CCompiler method)
set_seq1() (difflib.SequenceMatcher method)
status() (imap
       module)
statvfs() (in
       module
       subprocess)
STD_ERROR_HANDLE  
       (in module
       subprocess)
STD_INPUT_HANDLE  
       (in module
       subprocess)
STD_OUTPUT_HANDLE  
       (in module
       subprocess)
StdButtonBox (asyncio
       attribute)
stderr attribute)
stdin
stdout
stdin
attribute)
attribute)
STDOUT
stdout
attribute)
attribute)
attribute)
attribute)
set_seq2() (difflib.SequenceMatcher method) (subprocess.Popen attribute)
set_seqs() (difflib.SequenceMatcher method) (subprocess.TimeoutExpired attribute)
set_sequences() (mailbox.MH method) (mailbox.MHMessage method) (range attribute)
set_server_documentation() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
  (xmlrpc.server.DocXMLRPCServer method) (slice object)
set_server_name() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
  (xmlrpc.server.DocXMLRPCServer method) (slice object)
set_server_title() (xmlrpc.server.DocCGIXMLRPCRequestHandler method)
  (xmlrpc.server.DocXMLRPCServer method) (unittest.TestCase)
set_servername_callback (ssl.SSLContext attribute)
set_spacing() (formatter.formatter method) (step (pdb command))
set_start_method() (in module multiprocessing) (step (range attribute))
set_startup_hook() (in module readline) (step (slice object))
set_step() (bdb.Bdb method) (stop (pdb command))
set_subdir() (mailbox.MaildirMessage method) (stop (range attribute))
set_task_factory() (asyncio.loop method) (stop (slice object))
set_terminator() (asyncchat.async_chat method) (StopAsyncIteration exception)
set_threshold() (in module gc) (StopIteration exception)
set_trace() (bdb.Bdb method) (StopListening exception)
  (in module bdb) (StopTest() exception)
  (in module pdb) (StopTestRun exception)
set_trace_callback() (sqlite3.Connection method) (STORE_AC)
  (sqlite3.PyConnection method) (STORE_AT)
set_tunnel() (http.client.HTTPConnection method) (STORE_DEF)
  (http.client.HTTPSProxyConnection method) (STORE_FAS)
storbinary() (ftplib.FTP method) (STORE_GL)
store() (imaplib.IMAP4 method) (STORE_NA)
storlines() (ft
set_type() (email.message.Message method)
set_unittest_reportflags() (in module doctest)
set_unixfrom() (email.message.EmailMessage method)
set_until() (bdb.Bdb method)
set_url() (urllib.robotparser.RobotFileParser method)
set_usage() (optparse.OptionParser method)
set_userptr() (curses.panel.Panel method)
set_visible() (mailbox.BabylMessage method)
set_wakeup_fd() (in module signal)
set_write_buffer_limits() (asyncio.WriteTransport method)
setacl() (imaplib.IMAP4 method)
setannotation() (imaplib.IMAP4 method)
sattr() (built-in function)
setAttribute() (xml.dom.Element method)
setAttributeNode() (xml.dom.Element method)
setAttributeNodeNS() (xml.dom.Element method)
setAttributeNS() (xml.dom.Element method)
SetBase() (xml.parsers.expat.xmlparser method)
setblocking() (socket.socket method)
setByteStream() (xml.sax.xmlreader.InputSource method)
setbreak() (in module tty)
setCharacterStream() (xml.sax.xmlreader.InputSource method)
setcheckinterval() (in module sys)
setcomptype() (aifc.aifc method)
setcomptype() (sunau.AU_write method)
setcomptype() (wave.Wave_write method)
str (built-in class)
str() (in module string)
strcoll() (in module locale)
StreamError (class in module)
StreamHandler (class in logging)
StreamReader (class in asyncio)
StreamReaderWriter (class in codecs)
StreamRecoder (class in codecs)
StreamRequestHandler (class in socketserver)
streams
stackable
StreamWriter (class in codecs)
strerror (OSI)
strerror() (in module os)
strftime() (datetime.date method)
strftime() (datetime.datetime method)
strftime() (datetime.time method)
time
strict (csv.Dialect attribute)
strict_domain (http.cookiejar.DefaultCookiePolicy attribute)
strict_errors() (in module codecs)
strict_ns_domain (http.cookiejar.DefaultCookiePolicy attribute)
strict_ns_set_initial_dollar (http.cookiejar.DefaultCookiePolicy attribute)
setContentViewHandler()
(xml.sax.xmlreader.XMLReader method)
setcontext() (in module decimal)
setDaemon() (threading.Thread method)
setDefault() (dict method)
(http.cookies.Morsel method)
setDefaulttimeout() (in module socket)
setDopenflags() (in module sys)
setDocumentLocator()
(xml.sax.handler.ContentHandler method)
setDTDHandler()
(xml.sax.xmlreader.XMLReader method)
setegid() (in module os)
setEncoding() (xml.sax.xmlreader.InputSource method)
setEntityResolver()
(xml.sax.xmlreader.XMLReader method)
setErrorHandler()
(xml.sax.xmlreader.XMLReader method)
seteuid() (in module os)
setFeature() (xml.sax.xmlreader.XMLReader method)
setfirstweekday() (in module calendar)
setfmt() (ossaudiodev.oss_audio_device method)
setformatter() (logging.Handler method)
(setframerate() (aifc.aifc method)
(sunau.AU_write method)
(wave.Wave_write method)
setgid() (in module os)
setgroups() (in module os)
seth() (in module turtle)
setheading() (in module turtle)
sethostname() (in module socket)
SetInteger() (msilib.Record method)
setitem() (in module operator)
settimeout() (in module signal)
setLevel() (logging.Handler method)
    (logging.Logger method)
setlocale() (in module locale)
setLocale() (xml.sax.xmlreader.XMLReader method)
setLoggerClass() (in module logging)
setlogmask() (in module syslog)
setLogRecordFactory() (in module logging)
setMaxConns() (urllib.request.CacheFTPHandler method)
setmode() (in module msvcrt)
setName() (threading.Thread method)
setnchannels() (aifc.aifc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
setnframes() (aifc.aifc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
SetParamEntityParsing()
(xml.parsers.expat.xmlparser method)
setparameters() (ossaudiodev.oss_audio_device method)
setparams() (aifc.aifc method)
    (sunau.AU_write method)
    (wave.Wave_write method)
setpassword() (zipfile.ZipFile method)
setpgid() (in module os)
setpgid() (in module os)
setpos() (aifc.aifc method)
    (in module turtle)
(sunau.AU_read method)
(wave.Wave_read method)

setposition() (in module turtle)
setpriority() (in module os)
setprofile() (in module sys)

(in module threading)

SetProperty() (msilib.SummaryInformation method)

setProperty() (xml.sax.xmlreader.XMLReader method)

setPublicId() (xml.sax.xmlreader.InputSource method)

setquota() (imaplib.IMAP4 method)

setraw() (in module tty)

setrecursionlimit() (in module sys)

setregid() (in module os)

setresgid() (in module os)

setresuid() (in module os)

setreuid() (in module os)

setrlimit() (in module resource)

setsampwidth() (aifc.aifc method)

(sunau.AU_write method)
(wave.Wave_write method)

setsrcreg() (curses.window method)

setsid() (in module os)

setsockopt() (socket.socket method), [1], [2]

setstate() (codecs.IncrementalDecoder method)

(setstate() (codecs.IncrementalEncoder method)

(in module random)

setStream() (logging.StreamHandler method)

SetStream() (msilib.Record method)

SetString() (msilib.Record method)

setswitchinterval() (in module sys), [1]

(in module test.support)
setSystemId() (xml.sax.xmlreader.InputSource method)
setyx() (in module curses)
setTarget() (logging.handlers.MemoryHandler method)
settiltangle() (in module turtle)
settimeout() (socket.socket method)
setTimeout() (urllib.request.CacheFTPHandler method)
settrace() (in module sys)

(set in module threading)
setuid() (in module os)
setundobuffer() (in module turtle)
setup() (in module distutils.core)

(set in module turtle)
(subclass of BaseRequestHandler)
suite()

(suite) (in module parser)
suiteClass (unittest.TestLoader attribute)
supernet()

(supports subnets)
supernet_of()

(supports bytestrings)
supports_follow_symlinks

(supports directory forks)
supports_effective_ids

(supports file descriptors)
supports_fd

(supports unicode filenames)
supports_unichr

(set in module os.path)

(supports abstract operations)

(supports bytes operations)

(supports complex numbers)

(supports floating point numbers)

(supports integers)
setx() (in module turtle)
setzattr() (in module os)
sety() (in module turtle)
SF_APPEND (in module stat)
SF_ARCHIVED (in module stat)
SF_IMMUTABLE (in module stat)
SF_MNOWAIT (in module os)
SF_NODISKIO (in module os)
SF_NOUNLINK (in module stat)
SF_SNAPSHOT (in module stat)
SF.Sync (in module os)
Shape (class in turtle)
shape (memoryview attribute)
shape() (in module turtle)
shapesize() (in module turtle)
shapetransform() (in module turtle)
share() (socket.socket method)
shared_ciphers() (ssl.SSLSocket method)
shared_object_filename() (distutils.ccompiler.CCompiler method)
shearfactor() (in module turtle)
syncdown() (in module)
shorten() (in module textwrap)
shouldFlush()
(logging.handlers.BufferingHandler method)
(logging.handlers.MemoryHandler method)
shouldStop (unittest.TestResult attribute)
show() (curses.panel.Panel method)
show_code() (in module dis)
show_compilers() (in module distutils.ccompiler)
showsyntaxerror() (code.InteractiveInterpreter method)
showtraceback() (code.InteractiveInterpreter method)
showturtle() (in module turtle)
showwarning() (in module warnings)
shuffle() (in module random)
shutdown() (concurrent.futures.Executor method)
(imaplib.IMAP4 method)
(in module logging)
(multiprocessing.managers.BaseManager method)
(socket.socket method)
(socketserver.BaseServer method)
shutdown_asyncgens() (asyncio.loop method)
shutil (module)
side_effect (unittest.mock.Mock attribute)
SIG_BLOCK (in module signal)
SIG_DFL (in module signal)
SIG_IGN (in module signal)
SIG_SETMASK (in module signal)
SIG_UNBLOCK (in module signal)
SIGINT, [1]
siginterrupt() (in module signal)
signal

SyntaxError
SyntaxWarning
sys
(module)
sys (module)
sys.exc_info
sys.last_traceback
sys.meta_path
sys.modules
sys.path
sys.path_hooks
sys.path_importer_cache
sys.stderr
sys.stdin
sys.stdout
sys.exc (2to3 fixer)
sys.version
(sysconf (http.server.BaseHTTPRequestHandler attribute)
sysconf() (in module os)
sysconf_names (in module os)
sysconfig (module)
syslog (module)
syslog() (in module syslog)
SysLogHandler (class in logging.handlers)
system() (in module platform)
system_alias () (in module platform)
system_must_validate_cert () (in module test.support)
SystemError
(SystemExit, built-in)

module, [1]  
signal (module)  
signal() (in module signal)  
Signature (class in inspect)  
signature (inspect.BoundArguments attribute)  
signature() (in module inspect)  
sigpending() (in module signal)  
sigtimedwait() (in module signal)  
sigwait() (in module signal)  
sigwaitinfo() (in module signal)  

simple  
    statement  
Simple Mail Transfer Protocol  
SimpleCookie (class in http.cookies)  
simplefilter() (in module warnings)  
SimpleHandler (class in wsgiref.handlers)  
SimpleHTTPRequestHandler (class in http.server)  
SimpleNamespace (class in types)  
SimpleQueue (class in multiprocessing)  
    (class in queue)  
SimpleXMLRPCRequestHandler (class in xmlrpc.server)  
SimpleXMLRPCServer (class in xmlrpc.server)  
sin() (in module cmath)  
    (in module math)  

**single dispatch**  
SimpleAddressHeader (class in email.headerregistry)  
singledispatch() (in module functools)  

singleton  
    tuple  
sinh() (in module cmath)  
    (in module math)
SIO_KEEPALIVE_VALS (in module socket)
SIO_LOOPBACK_FAST_PATH (in module socket)
SIO_RCVALL (in module socket)
site (module)
site command line option
  --user-base
  --user-site
site-packages
directory
sitecustomize
module
sixtofour (ipaddress.IPv6Address attribute)
size (struct.Struct attribute)
  (tarfile.TarInfo attribute)
  (tracemalloc.Statistic attribute)
  (tracemalloc.StatisticDiff attribute)
  (tracemalloc.Trace attribute)
T

T_FMT (in module locale)
T_FMT_AMPM (in module locale)
tab
tab() (tkinter.ttk.Notebook method)
TabError
tabnanny (module)
tabs() (tkinter.ttk.Notebook method)
tabsize (textwrap.TextWrapper attribute)
tabular
data
tag (xml.etree.ElementTree.Element attribute)
tag_bind() (tkinter.ttk.Treeview method)
tag_configure() (tkinter.ttk.Treeview method)
tag_has() (tkinter.ttk.Treeview method)
tagName (xml.dom.Element attribute)
tail (xml.etree.ElementTree.Element attribute)
take_snapshot() (in module tracemalloc)
takewhile() (in module itertools)
tan() (in module cmath)
    (in module math)
tanh() (in module cmath)
    (in module math)
TarError
TarFile (class in tarfile), [1]
tarfile (module)

Tix
    tix_addbitmapdir() (tkinter.tix.tixCommand method)
tix_cget() (tkinter.tix.tixCommand method)
tix_configure() (tkinter.tix.tixCommand method)
tix_getbitmap() (tkinter.tix.tixCommand method)
tix_getimage() (tkinter.tix.tixCommand method)
tix_option_get() (tkinter.tix.tixCommand method)
tix_resetoptions() (tkinter.tix.tixCommand method)

Tk
    (class in tkinter)
    (class in tkinter.tix)

Tk Option Data Types
    TK_LIBRARY
Tkinter
tkinter (module)
tkinter.scrolledtext (module)
tkinter.tix (module)
tkinter.ttk (module)
TList (class in tkinter.tix)
TLS
    TLSv1 (ssl.TLSVersion)
    TLSv1_1 (ssl.TLSVersion)
tarfile command line option
   --create  <tarfile>  <source1>  ...  <sourceN>
   --extract  <tarfile>  [<output_dir>]
   --list  <tarfile>
   --test  <tarfile>
   -c  <tarfile>  <source1>  ...  <sourceN>
   -e  <tarfile>  [<output_dir>]
   -l  <tarfile>
   -t  <tarfile>
   -v,  --verbose

target
   deletion
   list,  [1]
   list assignment
   list,  deletion
   loop control

target
   (xml.dom.ProcessingInstruction attribute)
TarInfo (class in tarfile)
Task (class in asyncio)
task_done() (asyncio.Queue method)
   (multiprocessing.JoinableQueue method)
   (queue.Queue method)
tau (in module cmath)
   (in module math)
tb_frame (traceback attribute)
tb_lasti (traceback attribute)
tb_lineno (traceback attribute)
tb_locals  (unittest.TestResult attribute)

tLSv1_2  (ssl.TLSVersi
TLSv1_3  (ssl.TLSVersi
TLSVersion (class in ss
TMP
TMPDIR
to_bytes() (int method)
to_eng_string()  (decimal method)
   (decimal.Decimal method)
to_integral()  (decimal method)
to_integral_exact()  (decimal.Context method)
   (decimal.Decimal method)
to_integral_value()  (decimal.Decimal method)
to_sci_string()  (decimal.Context method)
ToASCII()  (in encodings.idna)
tobuf() (tarfile.TarInfo method)
tobytes() (array.array method)
   (memoryview method)
today()  (datetime.date class method)
   (datetime.datetime class method)
tofile() (array.array method)
tok_name (in module token)
   (module)
   (shlex.shlex attribute)
token_bytes() (in module token)
token_hex() (in module
tb_next (traceback attribute)
tbreak (pdb command)
tcdrain() (in module termios)
tcflow() (in module termios)
tcflush() (in module termios)
tcgetattr() (in module termios)
tcgetpgrp() (in module os)
Tcl() (in module tkinter)
TCL_LIBRARY
TCPServer (class in socketserver)
tcsendbreak() (in module termios)
tcsetattr() (in module termios)
tcsetpgrp() (in module os)
tearDown() (unittest.TestCase method)
tearDownClass() (unittest.TestCase method)
tee() (in module itertools)
tell() (aifc.aifc method), [1]
    (chunk.Chunk method)
    (io.IOBase method)
    (io.TextIOWrapper method)
    (mmap.mmap method)
    (sunau.AU_read method)
    (sunau.AU_write method)
    (wave.Wave_read method)
    (wave.Wave_write method)
Telnet (class in telnetlib)
telnetlib (module)
TEMP
temp_cwd() (in module test.support)
temp_dir() (in module test.support)
temp_umask() (in module test.support)
token_urlsafe() (in module secrets)
TokenError
tokenize (module)
tokenize command line
    -e, --exact
    -h, --help
tokenize() (in module tokenize)
tolist() (array.array method)
    (memoryview method)
    (parser.ST method)
tomono() (in module audioop)
toordinal() (datetime.date method)
    (datetime.datetime method)
top() (curses.panel.Panel method)
top_panel() (in module curses.panel)
toprettyxml() (xml.dom.minidom.Node method)
tostereo() (in module audioop)
tostring() (array.array method)
    (in xml.etree.ElementTree)
tostringlist() (in module xml.etree.ElementTree)
total_changes (sqlite3.Connection attribute)
total_ordering() (in module functools)
total_seconds() (datetime.timedelta method)
totuple() (parser.ST method)
touch() (pathlib.Path method)
touchline() (curses.window method)
tempdir (in module tempfile)
tempfile (module)
Template (class in pipes)
    (class in string)
template (string.Template attribute)
temporary
    file
    file name
TemporaryDirectory() (in module tempfile)
TemporaryFile() (in module tempfile)
teredo (ipaddress.IPv6Address attribute)
TERM, [1]
termattrs() (in module curses)
terminalseize (class in os)
terminate()
    (asyncio.asyncio.subprocess.Process method)
        (asyncio.SubprocessTransport method)
        (multiprocessing.Process method)
        (multiprocessing.pool.Pool method)
        (subprocess.Popen method)
termination model
termios (module)
terminename() (in module curses)
ternary
    operator
test
    identity
    membership
touchwin() (curses.window method)
touunicode() (array.array method)
ToUnicode() (in module encodings.idna)
towards() (in module turtle)
toxm1() (xml.dom.minidom method)
tp_as_async (C member)
tp_as_mapping (C member)
tp_as_number (C member)
tp_as_sequence (C member)
tparm() (in module curses)
trace
    stack
Trace (class in trace)
    (class in tracemalloc)
trace (module)
    trace command line option
        --help
        --ignore-dir=<dir>
        --ignore-module=<
        --version
        -C, --coverdir=<dir:
        -R, --no-report
        -T, --trackcalls
        -c, --count
        -f, --file=<file>
        -g, --timing
        -l, --listfuncs
        -m, --missing
        -r, --report
        -s, --summary
test (doctest.DocTestFailure attribute)
   (doctest.UnexpectedException attribute)
   (module)
test() (in module cgi)
test.support (module)
test.support.script_helper (module)
TEST_DATA_DIR (in module test.support)
TEST_HOME_DIR (in module test.support)
TEST_HTTP_URL (in module test.support)
TEST_SUPPORT_DIR (in module test.support)
TestCase (class in unittest)
TestFailed
testfile() (in module doctest)
TESTFN (in module test.support)
TESTFN_ENCODING (in module test.support)
TESTFN_NONASCII (in module test.support)
TESTFN_UNDECODABLE (in module test.support)
TESTFN_UNENCODABLE (in module test.support)
TESTFN_UNICODE (in module test.support)
TestHandler (class in test.support)
TestLoader (class in unittest)
testMethodPrefix (unittest.TestLoader attribute)
testmod() (in module doctest)
-t, --trace
trace function, [1], [2]
trace() (in module inspect)
trace_dispatch() (bdb.ExceptionTraceback object, [1], [2], [3], [4])
Traceback (class in traceback)
traceback (module)
traceback_dispatch() (tracemalloc.Statistic attribute)
traceback_limit (tracemalloc.Snapshot attribute)
TracebackException traceback)
tracebacklimit (in module)
tracebacks
   in CGI scripts
TracebackType (class in traceback)
tracemalloc (module)
tracer() (in module turtle)
traces (tracemalloc.Trace attribute)
trailing comma
transfercmd() (ftplib.FTP)
transientインターネット() (in module test.support)
TransientResource test.support)
translate() (bytearray method)
testNamePatterns
  (unittest.TestLoader attribute)
TestResult (class in unittest)
tests (in module imghdr)
testsource() (in module doctest)
testsRun (unittest.TestResult attribute)
TestSuite (class in unittest)
testzip() (zipfile.ZipFile method)
Text (class in typing)
text (in module msilib)
  (traceback.TracebackException attribute)
  (xml.etree.ElementTree.Element attribute)
text encoding
text file
text mode
text() (in module cgitb)
  (msilib.Dialog method)
text_factory (sqlite3.Connection attribute)
Textbox (class in curses.textpad)
TextCalendar (class in calendar)
textdomain() (in module gettext)
  (in module locale)
TextFile (class in distutils.text_file)
textinput() (in module turtle)
TextIO (class in typing)
TextIOWrapper (class in io)
TextTestResult (class in unittest)
TextTestRunner (class in unittest)
textwrap (module)
  (bytes method)
  (in module fnmatch)
  (str method)
translation() (in module gettext)
transport (asyncio.StreamWriter attribute)
Transport (class in asyncio)
Transport Layer Security
traverseproc (C type)
Tree (class in tkinter.tix)
TreeBuilder (xml.etree.ElementTree)
Treeview (class in tkinter.ttk)
triangular() (in module random)
triple-quoted string, [1]
True, [1], [2]
true
True (built-in variable)
truediv() (in module operator)
trunc() (in module math)
truncate() (in module os)
  (io.IOBase method)
truth
  value
truth() (in module operator)
  try
  statement, [1], [2]
ttk
tty
  I/O control
tty (module)
ttyname() (in module os)
tuple
  built-in function, [1]
TextWrapper (class in textwrap)

theme_create() (tkinter.ttk.Style method)

theme_names() (tkinter.ttk.Style method)

theme_settings() (tkinter.ttk.Style method)

theme_use() (tkinter.ttk.Style method)

THOUSEP (in module locale)

Thread (class in threading)

thread() (imaplib.IMAP4 method)

thread_info (in module sys)

thread_time() (in module time)

thread_time_ns() (in module time)

threading (module)

threading_cleanup() (in module test.support)

threading_setup() (in module test.support)

ThreadingHTTTPServer (class in http.server)

ThreadingMixIn (class in socketserver)

ThreadingTCPServer (class in socketserver)

ThreadingUDPServer (class in socketserver)

ThreadPoolExecutor (class in concurrent.futures)

threads

    POSIX

throw (2to3 fixer)

throw() (coroutine method)
   (generator method)

display empty, [1]

doctest object, [1], [2], [3],
singleton
tuple (built-in class)

Tuple (in module typing)
tuple2st() (in module parser)
tuple_params (2to3 fixer)

Turtle (class in turtle)
turtle (module)
turtledemo (module)
turtles() (in module turtle)

TurtleScreen (class in turtle)
turtlesize() (in module turtle)
type, [1]

    Boolean
    built-in function, [1]
    data
    hierarchy
    immutable data
    object, [1], [2]
    operations on dictionary
    operations on list

type (built-in class)

Type (class in typing)
type (optparse.Option attribute)
  (socket.socket attribute)
  (tarfile.TarInfo attribute)
  (urllib.request.Request attribute)

type alias
type hint
type of an object

TYPE_CHECKER
ticket_lifetime_hint  (ssl.SSLSession attribute)
tigetflag() (in module curses)
tigetnum() (in module curses)
tigetstr() (in module curses)
TILDE  (in module token)
tilt()  (in module turtle)
tiltangle() (in module turtle)
time  (class in datetime)
   (module)
   (ssl.SSLSession attribute)
time()  (asyncio.loop method)
   (datetime.datetime method)
   (in module time)
Time2Internaldate()  (in module imaplib)
time_ns()  (in module time)
timedelta  (class in datetime)
TimedRotatingFileHandler (class in logging.handlers)
timegm() (in module calendar)
timeit  (module)
timeit command line option
   -h, --help
   -n  N, --number=N
   -p, --process
   -r  N, --repeat=N
   -s  S, --setup=S
   -u, --unit=U
   -v, --verbose
timeit() (in module timeit)
   (timeit.Timer method)
timeout
   (socketserver.BaseServer
   (optparse.Option attribute)
TIME_CHECKING  (typing)
typeahead() (in module
   typecode (array.array attribute) a
typecodes (in module a
   TYPED_ACTIONS
   (optparse.Option attribute)
typed_subpart_iterator(email.iterators)
TypeError
   exception
types
   built-in
   immutable sequence
   module
   mutable sequence
   operations on integer
   operations on mapping
   operations on number
   operations on sequence
   types (2to3 fixer)
   (module)
   TYPES (optparse.Option attribute)
types, internal
   types_map (in module 
   (mimetypes.MimeTypes attribute)
types_map_inv
   (mimetypes.MimeTypes
TypeVar (class in typing)
typing (module)
TZ, [1], [2], [3], [4], [5]
tzinfo (class in datetime
attribute) (ssl.SSLSession attribute)
(subprocess.TimeoutExpired attribute)
timeout() (curses.window method)
TIMEOUT_MAX (in module _thread)
    (in module threading)
TimeoutError, [1], [2], [3]
TimeoutExpired
Timer (class in threading)
    (class in threading)
TimerHandle (class in asyncio)
times() (in module os)
TIMESTAMP
(py_compile.PycInvalidationMode attribute)
timestamp() (datetime.datetime method)
timetuple() (datetime.date method)
    (datetime.datetime method)
timetz() (datetime.datetime method)
timezone (class in datetime)
    (in module time)
title() (bytearray method)
    (bytes method)
    (in module turtle)
    (str method)
U

u"
  string literal
u'
  string literal
u-LAW, [1], [2]
ucd_3_2_0 (in module unicodedata)
udata (select.kevent attribute)
UDPServer (class in socketserver)
UF_APPEND (in module stat)
UF_COMPRESSED (in module stat)
UF_HIDDEN (in module stat)
UF_IMMUTABLE (in module stat)
UF_NODUMP (in module stat)
UF_NOUNLINK (in module stat)
UF_OPAQUE (in module stat)
uid (tarfile.TarInfo attribute)
uid() (imaplib.IMAP4 method)
uidl() (poplib.POP3 method)
ulaw2lin() (in module audioop)
ULONG_MAX
umask() (in module os)
unalias (pdb command)
uname (tarfile.TarInfo attribute)
uname() (in module os)
  (in module platform)
unary
  arithmetic operation
  bitwise operation
UNARY_INVERT (opcode)
UNARY_NEGATIVE (opcode)
UNARY_NOT (opcode)
UNARY_POSITIVE (opcode)
unparsedEntityDecl()
  (xml.sax.handler.DTDHandler method)
UnparsedEntityDeclHandler()
  (xml.parsers.expat.xmlparser method)
Unpickler (class in pickle)
UnpicklingError
unquote() (in module urllib.parse)
unquote_plus() (in module urllib.parse)
unquote_to_bytes() (in module urllib.parse)
unreachable object
unreadline() (distutils.text_file.TextFile method)
unrecognized escape sequence
unregister() (in module atexit)
  (in module faulthandler)
  (select.devpoll method)
  (select.epoll method)
  (select.poll method)
  (selectors.BaseSelector method)
unregister_archive_format() (in module shutil)
unregister_dialect() (in module csv)
unregister_unpack_format() (in module shutil)
unsafe (uuid.SafeUUID attribute)
unset() (test.support.EnvironmentVarGuard method)
unsetenv() (in module os)
UnstructuredHeader (class in email.headerregistry)
unsubscribe() (imaplib.IMAP4 method)
UnsupportedOperation
until (pdb command)
untokenize() (in module tokenize)
unbinding
   name
UnboundLocalError, [1]
unbuffered I/O
UNC paths
   and os.makedirs()
UNCHECKED_HASH
   (py_compile.PycInvalidationMode attribute)
unconsumed_tail
   (zlib.Decompress attribute)
unctrl() (in module curses)
   (in module curses.ascii)
undefine_macro()
   (distutils.ccompiler.CCompiler method)
Underflow (class in decimal)
undisplay (pdb command)
undo() (in module turtle)
undobufferentries() (in module turtle)
undoc_header (cmd.Cmd attribute)
unescape() (in module html)
   (in module xml.sax.saxutils)
UnexpectedException
unexpectedSuccesses (unittest.TestResult attribute)
unfreeze() (in module gc)
uget_wch() (in module curses)
ugetch() (in module curses)
   (in module msvcrtd)
ugetmouse() (in module curses)
ugetwch() (in module msvcrtd)
unhexlify() (in module binascii)
Unicode, [1], [2]
   database
unicode (2to3 fixer)
untouchwin() (curses.window method)
unused_data (bz2.BZ2Decompressor attribute)
unverifiable (urllib.request.Request attribute)
unwrap() (in module functools)
up (pdb command)
up() (in module turtle)
update() (collections.Counter method)
dict method
frozenset method
hashlib.hash method
hmac.HMAC method
http.cookies.Morsel method
mailbox.Mailbox method
mailbox.Maildir method
trace.CoverageResults method
update_authenticated (urllib.request.HTTPPasswordMgrWithPriorAuth method)
update_lines_cols()
update_panels()
update_visible()
update_wrapper()
upper() (bytes method)
str method
urandom() (in module os)
URL, [1], [2], [3]
parsing
relative
Unicode Consortium
unicodedata (module)
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
UnicodeWarning
unidata_version (in module unicodedata)
unified_diff() (in module difflib)
uniform() (in module random)
UnimplementedFileMode
Union (class in ctypes)
    (in module typing)
union() (frozenset method)
unique() (in module enum), [1]
unittest (module)
unitest command line option
    --locals
    -b, --buffer
    -c, --catch
    -f, --failfast
    -k
unitest-discover command line option
    -p, --pattern pattern
    -s, --start-directory directory
    -t, --top-level-directory directory
    -v, --verbose
unittest.mock (module)
universal newlines
    What's new, [1], [2], [3]
bytearray.splitlines method
bytes.splitlines method
csv.reader function
importlib.abc.InspectLoader.get_source
url (xmlrpc.client.
url2pathname() (i
urlcleanup() (in m
urldefrag() (in m
urlencode() (in m
URLLError
urljoin() (in modu
urllib (2to3 fixer)
    (module)
urllib.error (modu
urllib.parse (modu
urllib.request
    module
urllib.request (mc
urllib.response (n
urllib.robotparser
urlopen() (in mod
URLopener (clas
urllibparse() (in mod
urlretrieve() (in m
urllibsafe_b64decod
urllibsafe_b64encod
urlsplit() (in modu
urllunparse() (in n
urlunsplit() (in m
urn (uuid.UUID a
use_default_color
use_env() (in mo
use_rawinput (cn
UseForeignDTD(
method)
USER
user
effective id
method
io.IncrementalNewlineDecoder class
io.TextIOWrapper class
open() built-in function
str.splitlines method
subprocess module
UNIX
  I/O control
  file control
unix_dialect (class in csv)
unix_shell (in module test.support)
UnixDatagramServer (class in socketserver)
UnixStreamServer (class in socketserver)
unknown (uuid.SafeUUID attribute)
unknown_decl() (html.parser.HTMLParser method)
unknown_open()
(urllib.request.BaseHandler method)
  (urllib.request.UnknownHandler method)
UnknownHandler (class in urllib.request)
UnknownProtocol
UnknownTransferEncoding
unlink() (in module os)
  (in module test.support)
  (pathlib.Path method)
  (xml.dom.minidom.Node method)
unload() (in module test.support)
unlock() (mailbox.Babyl method)
  (mailbox.MH method)
  (mailbox.MMDF method)
  (mailbox.Mailbox method)
  (mailbox.Maildir method)

id
  id, setting
user() (poplib.POP3 method)
user-defined function
user-defined function call
method
user-defined func
object, [1], [2]
USER_BASE
  (in module site)
user_call() (bdb.Bdb method)
user_exception() (bdb.Bdb method)
user_line() (bdb.Bdb method)
user_return() (bd USER_SITE (in r
usercustomize
  module
UserDict (class in collections)
UserList (class in collections)
USERNAME, [1]
username
attribute)
USERPROFILE,
userptr() (curses.
UserString (class
UserWarning
USTAR_FORMAT
UTC
utc (datetime.timedelta
utcfromtimestamp method)
unpack() (in module struct)
  (struct.Struct method)
unpack_archive() (in module shutil)
unpack_array() (xdrlib.Unpacker method)
unpack_bytes() (xdrlib.Unpacker method)
unpack_double() (xdrlib.Unpacker method)
UNPACK_EX (opcode)
unpack_farray() (xdrlib.Unpacker method)
unpack_float() (xdrlib.Unpacker method)
unpack_fopaque() (xdrlib.Unpacker method)
unpack_from() (in module struct)
  (struct.Struct method)
unpack_fstring() (xdrlib.Unpacker method)
unpack_list() (xdrlib.Unpacker method)
unpack_opaque() (xdrlib.Unpacker method)
UNPACK_SEQUENCE (opcode)
unpack_string() (xdrlib.Unpacker method)
Unpacker (class in xdrlib)
unpacking
  dictionary
  in function calls
  iterable
v4_int_to_packed() (in module ipaddress)
v6_int_to_packed() (in module ipaddress)
validator() (in module wsgiref.validate)
value
default parameter
truth
value (ctypes._SimpleCData attribute)
(http.cookiejar.Cookie attribute)
(http.cookies.Morsel attribute)
(xml.dom.Attr attribute)
value of an object
Value() (in module multiprocessing)
(in module multiprocessing.sharedctypes)
(multiprocessing.managers.SyncManager method)
value_decode() (http.cookies.BaseCookie method)
value_encode() (http.cookies.BaseCookie method)
ValueError
exception
valuerefs() (weakref.WeakValueDictionary method)
values
Boolean
writing
values() (contextvars.Context method)
(dict method)
(email.message.EmailMessage method)
(email.message.Message method)
(mailbox.Mailbox method)
(types.MappingProxyType method)
verify() (smtplib.SMTP method)
verify_client_post_handshake() (ssl.SSLSocket method)
verify_code (ssl.SSLCertVerificationError attribute)
VERIFY_CRL_CHECK_CHAIN (in module ssl)
VERIFY_CRL_CHECK_LEAF (in module ssl)
VERIFY_DEFAULT (in module ssl)
verify_flags (ssl.SSLCertVerificationError attribute)
verify_mode (ssl.SSLContext attribute)
verify_request() (socketserver.BaseServer method)
VERIFY_X509 (module ssl)
VerifyFlags (class in ssl)
VerifyMode (class in ssl)
version
(email.headerregistry.MIMEVersionHeader attribute)
http.client.HTTPResponse attribute)
http.cookiejar.Cookie attribute)
in module curses
(in module marshal
(in module sqlite3
(in module sys
(ipaddress.IPv4Address attribute)
(ipaddress.IPv4Network attribute)
ValuesView (class in collections.abc)
    (class in typing)
var   (contextvars.contextvars.Token.Token attribute)
    variable
    free
    variable annotation
    variance() (in module statistics)
    variant (uuid.UUID attribute)
    vars() (built-in function)
    VBAR (in module token)
    vbar    (tkinter.scrolledtext.ScrolledText attribute)
    VBAREQUAL (in module token)
    Vec2D (class in turtle)
    venv (module)
    VERBOSE (in module re)
    verbose (in module tabnanny)
    (in module test.support)
    (ipaddress
        (ipaddress
    (urllib.request
        (uuid.UUID
    version() (in module
        (ssl.SSLSocket
    version_info (in
    version_string(
    vformat() (string
    virtual
    Environment
    virtual environment
    virtual machine
    visit() (ast.Node
    visitproc (C type
    vline() (curses.
    voidcmd() (ftplib.
    volume (zipfile.
    vonmisesvariat
W

W_OK (in module os)
wait()

(asyncio.asyncio.subprocess.Process method)

(asyncio.Condition method)

(asyncio.Event method)

(in module asyncio)

(in module concurrent.futures)

(in module multiprocessing.connection)

(in module os)

(multiprocessing.pool.AsyncResult method)

(subprocess.Popen method)

(threading.Barrier method)

(threading.Condition method)

(threading.Event method)

wait3() (in module os)

wait4() (in module os)

wait_closed() (asyncio.Server method)

(asyncio.StreamWriter method)

wait_for() (asyncio.Condition method)

(in module asyncio)

(threading.Condition method)

wait_threads_exit() (in module test.support)

waitid() (in module os)

waitpid() (in module os)

walk() (email.message.EmailMessage method)

(email.message.Message method)

(in module ast)

with_name() (pathlib.PurePath method)

with_netmask (ipaddress.IPv4Network attribute)

(with_prefixlen (ipaddress.IPv6Network attribute)

(multiprocessing.connection)

(multiprocessing.pool.AsyncResult method)

(subprocess.Popen method)

(threading.Barrier method)

(threading.Condition method)

(threading.Event method)

with_pymalloc() (in module test.support)

with_suffix() (pathlib.PurePath method)

with_traceback() (BaseException method)

WNOHANG (in module os)

WNOWAIT (in module os)

wordchars (shlex.shlex attribute)

World Wide Web, [1],

wrap() (in module textwrap)

(textwrap.TextWrapper method)

wrap_bio() (ssl.SSLContext method)

wrap_future() (in module asyncio)

wrap_socket() (in module ssl)

wrap_text() (distutils.fancy_getopt)

wrapper() (in module functools)

WrapperDescriptorType (in module types)

wraps() (in module functools)

WRITABLE (in module tkinter)
(in module os)
walk_packages() (in module pkgutil)
walk_stack() (in module traceback)
walk_tb() (in module traceback)
want (doctest.Example attribute)
warn() (distutils.ccompiler.CCompiler method)
    (distutils.text_file.TextFile method)
    (in module warnings)
warn_explicit() (in module warnings)
Warning, [1]
warning() (in module logging)
    (logging.Logger method)
    (xml.sax.handler.ErrorHandler method)
warnings
    (module)
WarningsRecorder (class in test.support)
warnoptions (in module sys)
wasSuccessful() (unittest.TestResult method)
WatchedFileHandler (class in logging.handlers)
wave (module)
WCONTINUED (in module os)
WCOREDUMP() (in module os)
WeakKeyDictionary (class in weakref)
WeakMethod (class in weakref)
weakref (module)
WeakSet (class in weakref)
WeakValueDictionary (class in weakref)
webbrowser (module)

writable() (asyncore.d
    (io.IOBase method)
write() (asyncio.StreamWriter method)
    (asyncio.WriteTransport method)
    (code.InteractiveInterpreter method)
    (codecs.StreamWriter method)
    (configparser.ConfigParser method)
    (email.generator.Generator method)
    (email.generator.Message method)
    (in module os)
    (in module turtle)
    (io.BufferedIOBase method)
    (io.BufferedWriter method)
    (io.RawIOBase method)
    (io.TextIOBase method)
    (mmap.mmap method)
    (ossaudiodev.ossaudiodev method)
    (ssl.MemoryBIO method)
    (ssl.SSLSocket method)
    (telnetlib.Telnet method)
    (xml.etree.ElementTree method)
    (zipfile.ZipFile method)
write_byte() (mmap.mmap method)
write_bytes() (pathlib.Path method)
write_docstringdict() (ast method)
write_eof() (asyncio.WriteTransport method)
write_file() (in module)
write_history_file() (in
weekday() (datetime.date method)  
  (datetime.datetime method)  
  (in module calendar)  
weekheader() (in module calendar)  
weibullvariate() (in module random)  
WEXITED (in module os)  
WEXITSTATUS() (in module os)  
wfile  
  (http.server.BaseHTTPRequestHandler attribute)  
what() (in module imghdr)  
  (in module sndhdr)  
whathdr() (in module sndhdr)  
whatis (pdb command)  
when() (asyncio.TimerHandle method)  
where (pdb command)  
which() (in module shutil)  
whichdb() (in module dbm)  
while  
  statement, [1], [2], [3]  
whitespace (in module string)  
  (shlex.shlex attribute)  
whitespace_split (shlex.shlex attribute)  
Widget (class in tkinter.ttk)  
width (textwrap.TextWrapper attribute)  
width() (in module turtle)  
WIFCONTINUED() (in module os)  
WIFEXITED() (in module os)  
WIFSIGNALED() (in module os)  
WIFSTOPPED() (in module os)  
win32_ver() (in module platform)  
WinDLL (class in ctypes)  
window manager (widgets)  
window() (curses.panel.Panel method)  
WRITE_RESTRICTED  
write_results() (traceback method)  
write_text() (pathlib.Path attribute)  
write_through  
writeall() (ossaudiodev method)  
writeframes() (aifc.aifc method)  
  (sunau.AU_write method)  
  (wave.Wave_write method)  
writeframesraw() (aifc.aifc method)  
  (sunau.AU_write method)  
  (wave.Wave_write method)  
writeheader() (csv.DictWriter method)  
writelines() (asyncio.StreamWriter method)  
  (asyncio.WriteTransport method)  
  (codecs.StreamWriter method)  
  (io.IOBase method)  
writePlist() (in module plistlib)  
writePlistToBytes() (in module plistlib)  
writepy() (zipfile.PyZipFile method)  
writer (formatter.formatter attribute)  
writer() (in module csv)  
writerow() (csv.csvwriter method)  
writerows() (csv.csvwriter method)  
writestr() (zipfile.ZipFile method)  
WriteTransport (class in asyncio)  
writev() (in module os)  
writexml() (xml.dom.minidom.Node method)  
writing  
  values
window_height() (in module turtle)
window_width() (in module turtle)
Windows
Windows ini file
WindowsError
WindowsPath (class in pathlib)
WindowsProactorEventLoopPolicy (class in asyncio)
WindowsRegistryFinder (class in importlib.machinery)
winerror (OSError attribute)
WinError() (in module ctypes)
WINFUNCTYPE() (in module ctypes)
winreg (module)
WinSock
winsound (module)
winver (in module sys)
with
    statement, [1]
    WITH_CLEANUP_FINISH (opcode)
    WITH_CLEANUP_START (opcode)
with_hostmask (ipaddress.IPv4Interface attribute)
    (ipaddress.IPv4Network attribute)
    (ipaddress.IPv6Interface attribute)
    (ipaddress.IPv6Network attribute)
WrongDocumentErr
ws_comma (2to3 fixer)
wsgi_file_wrapper (wsgiref.handlers.Base)
wsgi_multiprocess (wsgiref.handlers.Base)
wsgi_multithread (wsgiref.handlers.Base)
wsgi_run_once (wsgiref.handlers.Base)
wsgiref (module)
wsgiref.handlers (module)
wsgiref.headers (module)
wsgiref.simple_server
wsgiref.util (module)
wsgiref.validate (module)
WSGIRequestHandler (in wsgiref.simple_server)
WSGIServer (in wsgiref.simple_server)
WShowWindow (subprocess.STARTUP)
WSTOPPED (in module)
WSTOPSIG() (in module)
wstring_at() (in module)
WTERMSIG() (in module)
WUNTRACED (in module)
WWW, [1], [2]
    server, [1]
X

X (in module re)
X509 certificate
X_OK (in module os)
xatom() (imaplib.IMAP4 method)
XATTR_CREATE (in module os)
XATTR_REPLACE (in module os)
XATTR_SIZE_MAX (in module os)
xcor() (in module turtle)
XDR
xdrlib (module)
xhdr() (nntplib.NNTP method)
XHTML
XHTML_NAMESPACE (in module xml.dom)
xml (module)
XML() (in module xml.etree.ElementTree)
xml.dom (module)
xml.dom.minidom (module)
xml.dom.pulldom (module)
xml.etree.ElementTree (module)
xml.parsers.expat (module)
xml.parsers.expat.errors (module)
xml.parsers.expat.model (module)
xml.sax (module)
xml.sax.handler (module)
xml.sax.saxutils (module)
xml.sax.xmlreader (module)
XML_ERROR_ABORTED (in module xml.parsers.expat.errors)
XML_ERROR_ASYNC_ENTITY (in module xml.parsers.expat.errors)
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (in module xml.parsers.expat.errors)
Y

Y2K
ycor() (in module turtle)
year (datetime.date attribute)
   (datetime.datetime
    attribute)
Year 2000
Year 2038
yeardatescalendar()
(calendar.Calendar method)
yeardays2calendar()
(calendar.Calendar method)
yeardayscalendar()
(calendar.Calendar method)
YESEXPR (in module locale)

yield
eexamples
eexpression
ekkeyword
estatement
vyield from (in What's
   New)
YIELD_FROM (opcode)
YIELD_VALUE (opcode)
yiq_to_rgb() (in module
colorsys)
yview() (tkinter.ttk.Treeview
   method)
Zen of Python
ZeroDivisionError exception
zfill() (bytearray method)
  (bytes method)
  (str method)
zip (2to3 fixer)
zip() (built-in function)
ZIP_BZIP2 (in module zipfile)
ZIP_DEFLATED (in module zipfile)
zip_longest() (in module itertools)
ZIP_LZMA (in module zipfile)
ZIP_STORED (in module zipfile)
zipapp (module)
zipapp command line option
  --info
  -c, --compress
  -h, --help
  -m <mainfn>, --main=<mainfn>
  -o <output>, --output=<output>
  -p <interpreter>, --python=<interpreter>
ZipFile (class in zipfile)
zipfile (module)
zipfile command line option
  --create <zipfile> <source1> ...
  --extract <zipfile>
  <output_dir>
  --list <zipfile>
  --test <zipfile>
  -c <zipfile> <source1> ...
  <sourceN>
  -e <zipfile> <output_dir>
  -l <zipfile>
  -t <zipfile>
zipimport (module)
zipimporter (class in zipimport)
ZipImportError
ZipInfo (class in zipfile)
zlib (module)
ZLIB_RUNTIME_VERSION (in module zlib)
ZLIB_VERSION (in module zlib)
9. Command Reference
9.1. Installing modules: the **install** command family

The install command ensures that the build commands have been run and then runs the subcommands **install_lib**, **install_data** and **install_scripts**.

9.1.1. **install_data**

This command installs all data files provided with the distribution.

9.1.2. **install_scripts**

This command installs all (Python) scripts in the distribution.
9.2. Creating a source distribution: the \texttt{sdist} command

The manifest template commands are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{include pat1 pat2 ...}</td>
<td>include all files matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{exclude pat1 pat2 ...}</td>
<td>exclude all files matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{recursive-include dir pat1 pat2 ...}</td>
<td>include all files under \texttt{dir} matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{recursive-exclude dir pat1 pat2 ...}</td>
<td>exclude all files under \texttt{dir} matching any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{global-include pat1 pat2 ...}</td>
<td>include all files anywhere in the source tree matching — &amp; any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{global-exclude pat1 pat2 ...}</td>
<td>exclude all files anywhere in the source tree matching — &amp; any of the listed patterns</td>
</tr>
<tr>
<td>\texttt{prune dir}</td>
<td>exclude all files under \texttt{dir}</td>
</tr>
<tr>
<td>\texttt{graft dir}</td>
<td>include all files under \texttt{dir}</td>
</tr>
</tbody>
</table>

The patterns here are Unix-style “glob” patterns: \texttt{*} matches any sequence of regular filename characters, \texttt{?} matches any single regular filename character, and \texttt{[range]} matches any of the characters in \texttt{range} (e.g., \texttt{a-z}, \texttt{a-zA-Z}, \texttt{a-f0-9_}). The definition of “regular filename character” is platform-specific: on Unix it is anything except
slash; on Windows anything except backslash or colon.
3. Writing the Setup Configuration File

Often, it’s not possible to write down everything needed to build a distribution a priori: you may need to get some information from the user, or from the user’s system, in order to proceed. As long as that information is fairly simple—a list of directories to search for C header files or libraries, for example—then providing a configuration file, setup.cfg, for users to edit is a cheap and easy way to solicit it. Configuration files also let you provide default values for any command option, which the installer can then override either on the command-line or by editing the config file.

The setup configuration file is a useful middle-ground between the setup script—which, ideally, would be opaque to installers [1]—and the command-line to the setup script, which is outside of your control and entirely up to the installer. In fact, setup.cfg (and any other Distutils configuration files present on the target system) are processed after the contents of the setup script, but before the command-line. This has several useful consequences:

- installers can override some of what you put in setup.py by editing setup.cfg
- you can provide non-standard defaults for options that are not easily set in setup.py
- installers can override anything in setup.cfg using the command-line options to setup.py

The basic syntax of the configuration file is simple:

```
[command]
```
where *command* is one of the Distutils commands (e.g. *build_py*, *install*), and *option* is one of the options that command supports. Any number of options can be supplied for each command, and any number of command sections can be included in the file. Blank lines are ignored, as are comments, which run from a '#' character until the end of the line. Long option values can be split across multiple lines simply by indenting the continuation lines.

You can find out the list of options supported by a particular command with the universal --help option, e.g.

```
$ python setup.py --help build_ext
[...]
Options for 'build_ext' command:
 --build-lib (-b) directory for compiled extension
 --build-temp (-t) directory for temporary files
 --inplace (-i) ignore build-lib and put compiled extension directly in source directory alongside your pure Python modules
 --include-dirs (-I) list of directories to search for header files
 --define (-D) C preprocessor macros to define
 --undef (-U) C preprocessor macros to undefine
 --swig-opts list of SWIG command line options
[...]
```

Note that an option spelled --foo-bar on the command-line is spelled `foo_bar` in configuration files.

For example, say you want your extensions to be built “in-place”—that is, you have an extension pkg.ext, and you want the compiled extension file (ext.so on Unix, say) to be put in the same source
directory as your pure Python modules pkg.mod1 and pkg.mod2. You can always use the --inplace option on the command-line to ensure this:

```
python setup.py build_ext --inplace
```

But this requires that you always specify the **build_ext** command explicitly, and remember to provide --inplace. An easier way is to “set and forget” this option, by encoding it in `setup.cfg`, the configuration file for this distribution:

```
[build_ext]
inplace=1
```

This will affect all builds of this module distribution, whether or not you explicitly specify **build_ext**. If you include `setup.cfg` in your source distribution, it will also affect end-user builds—which is probably a bad idea for this option, since always building extensions in-place would break installation of the module distribution. In certain peculiar cases, though, modules are built right in their installation directory, so this is conceivably a useful ability. (Distributing extensions that expect to be built in their installation directory is almost always a bad idea, though.)

Another example: certain commands take a lot of options that don’t change from run to run; for example, `bdist_rpm` needs to know everything required to generate a “spec” file for creating an RPM distribution. Some of this information comes from the setup script, and some is automatically generated by the Distutils (such as the list of files installed). But some of it has to be supplied as options to `bdist_rpm`, which would be very tedious to do on the command-line for every run. Hence, here is a snippet from the Distutils’ own `setup.cfg`:
[bdist_rpm]
release = 1
packager = Greg Ward <gward@python.net>
doc_files = CHANGES.txt
       README.txt
       USAGE.txt
doc/
       examples/

Note that the `doc_files` option is simply a whitespace-separated string split across multiple lines for readability.

See also:

**Syntax of config files** in “Installing Python Modules”

More information on the configuration files is available in the manual for system administrators.

Footnotes

[1] This ideal probably won’t be achieved until auto-configuration is fully supported by the Distutils.
2. Writing the Setup Script

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing. As we saw in section A Simple Example above, the setup script consists mainly of a call to setup(), and most information supplied to the Distutils by the module developer is supplied as keyword arguments to setup().

Here’s a slightly more involved example, which we’ll follow for the next couple of sections: the Distutils’ own setup script. (Keep in mind that although the Distutils are included with Python 1.6 and later, they also have an independent existence so that Python 1.5.2 users can use them to install other module distributions. The Distutils’ own setup script, shown here, is used to install the package into Python 1.5.2.)

```python
#!/usr/bin/env python

from distutils.core import setup

setup(name='Distutils',
    version='1.0',
    description='Python Distribution Utilities',
    author='Greg Ward',
    author_email='gward@python.net',
    url='https://www.python.org/sigs/distutils-sig/',
    packages=['distutils', 'distutils.command'],
    )
```

There are only two differences between this and the trivial one-file
distribution presented in section A Simple Example: more metadata, and the specification of pure Python modules by package, rather than by module. This is important since the Distutils consist of a couple of dozen modules split into (so far) two packages; an explicit list of every module would be tedious to generate and difficult to maintain. For more information on the additional meta-data, see section Additional meta-data.

Note that any pathnames (files or directories) supplied in the setup script should be written using the Unix convention, i.e. slash-separated. The Distutils will take care of converting this platform-neutral representation into whatever is appropriate on your current platform before actually using the pathname. This makes your setup script portable across operating systems, which of course is one of the major goals of the Distutils. In this spirit, all pathnames in this document are slash-separated.

This, of course, only applies to pathnames given to Distutils functions. If you, for example, use standard Python functions such as `glob.glob()` or `os.listdir()` to specify files, you should be careful to write portable code instead of hardcoding path separators:

```python
import os
import glob

glob.glob(os.path.join('mydir', 'subdir', '*.html'))
o.listdir(os.path.join('mydir', 'subdir'))
```
2.1. Listing whole packages

The packages option tells the Distutils to process (build, distribute, install, etc.) all pure Python modules found in each package mentioned in the packages list. In order to do this, of course, there has to be a correspondence between package names and directories in the filesystem. The default correspondence is the most obvious one, i.e. package distutils is found in the directory distutils relative to the distribution root. Thus, when you say packages = ['foo'] in your setup script, you are promising that the Distutils will find a file foo/__init__.py (which might be spelled differently on your system, but you get the idea) relative to the directory where your setup script lives. If you break this promise, the Distutils will issue a warning but still process the broken package anyway.

If you use a different convention to lay out your source directory, that's no problem: you just have to supply the package_dir option to tell the Distutils about your convention. For example, say you keep all Python source under lib, so that modules in the “root package” (i.e., not in any package at all) are in lib, modules in the foo package are in lib/foo, and so forth. Then you would put

```
package_dir = {'': 'lib'}
```

in your setup script. The keys to this dictionary are package names, and an empty package name stands for the root package. The values are directory names relative to your distribution root. In this case, when you say packages = ['foo'], you are promising that the file lib/foo/__init__.py exists.

Another possible convention is to put the foo package right in lib,
the foo.bar package in lib/bar, etc. This would be written in the setup script as

```
package_dir = {'foo': 'lib'}
```

A package: dir entry in the package_dir dictionary implicitly applies to all packages below package, so the foo.bar case is automatically handled here. In this example, having packages = ['foo', 'foo.bar'] tells the Distutils to look for lib/__init__.py and lib/bar/__init__.py. (Keep in mind that although package_dir applies recursively, you must explicitly list all packages in packages: the Distutils will not recursively scan your source tree looking for any directory with an __init__.py file.)
2.2. Listing individual modules

For a small module distribution, you might prefer to list all modules rather than listing packages—especially the case of a single module that goes in the “root package” (i.e., no package at all). This simplest case was shown in section A Simple Example; here is a slightly more involved example:

```python
py_modules = ['mod1', 'pkg.mod2']
```

This describes two modules, one of them in the “root” package, the other in the pkg package. Again, the default package/directory layout implies that these two modules can be found in `mod1.py` and `pkg/mod2.py`, and that `pkg/__init__.py` exists as well. And again, you can override the package/directory correspondence using the `package_dir` option.
2.3. Describing extension modules

Just as writing Python extension modules is a bit more complicated than writing pure Python modules, describing them to the Distutils is a bit more complicated. Unlike pure modules, it's not enough just to list modules or packages and expect the Distutils to go out and find the right files; you have to specify the extension name, source file(s), and any compile/link requirements (include directories, libraries to link with, etc.).

All of this is done through another keyword argument to setup(), the `ext_modules` option. `ext_modules` is just a list of `Extension` instances, each of which describes a single extension module. Suppose your distribution includes a single extension, called `foo` and implemented by `foo.c`. If no additional instructions to the compiler/linker are needed, describing this extension is quite simple:

```python
Extension('foo', ['foo.c'])
```

The Extension class can be imported from `distutils.core` along with setup(). Thus, the setup script for a module distribution that contains only this one extension and nothing else might be:

```python
from distutils.core import setup, Extension
setup(name='foo',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
)
```

The Extension class (actually, the underlying extension-building machinery implemented by the `build_ext` command) supports a great deal of flexibility in describing Python extensions, which is explained in
the following sections.

2.3.1. Extension names and packages

The first argument to the `Extension` constructor is always the name of the extension, including any package names. For example,

```
Extension('foo', ['src/foo1.c', 'src/foo2.c'])
```

describes an extension that lives in the root package, while

```
Extension('pkg.foo', ['src/foo1.c', 'src/foo2.c'])
```

describes the same extension in the `pkg` package. The source files and resulting object code are identical in both cases; the only difference is where in the filesystem (and therefore where in Python’s namespace hierarchy) the resulting extension lives.

If you have a number of extensions all in the same package (or all under the same base package), use the `ext_package` keyword argument to `setup()`. For example,

```
setup(...,
    ext_package='pkg',
    ext_modules=[Extension('foo', ['foo.c']),
                 Extension('subpkg.bar', ['bar.c'])
    ]
)
```

will compile `foo.c` to the extension `pkg.foo`, and `bar.c` to `pkg.subpkg.bar`.

2.3.2. Extension source files
The second argument to the `Extension` constructor is a list of source files. Since the Distutils currently only support C, C++, and Objective-C extensions, these are normally C/C++/Objective-C source files. (Be sure to use appropriate extensions to distinguish C++ source files: `.cc` and `.cpp` seem to be recognized by both Unix and Windows compilers.)

However, you can also include SWIG interface (.i) files in the list; the `build_ext` command knows how to deal with SWIG extensions: it will run SWIG on the interface file and compile the resulting C/C++ file into your extension.

This warning notwithstanding, options to SWIG can be currently passed like this:

```python
setup(...,
    ext_modules=[
        Extension('_foo', ['foo.i'],
            swig_opt=['-modern', ''],
        )

    py_modules=['foo'],
)
```

Or on the commandline like this:

```bash
> python setup.py build_ext --swig-opts="-modern -I../include"
```

On some platforms, you can include non-source files that are processed by the compiler and included in your extension. Currently, this just means Windows message text (.mc) files and resource definition (.rc) files for Visual C++. These will be compiled to binary resource (.res) files and linked into the executable.

### 2.3.3. Preprocessor options
Three optional arguments to `Extension` will help if you need to specify include directories to search or preprocessor macros to define/undefine: `include_dirs`, `define_macros`, and `undef_macros`.

For example, if your extension requires header files in the `include` directory under your distribution root, use the `include_dirs` option:

```python
Extension('foo', ['foo.c'], include_dirs=['include'])
```

You can specify absolute directories there; if you know that your extension will only be built on Unix systems with X11R6 installed to `/usr`, you can get away with

```python
Extension('foo', ['foo.c'], include_dirs=['/usr/include'])
```

You should avoid this sort of non-portable usage if you plan to distribute your code: it's probably better to write C code like

```c
#include <X11/Xlib.h>
```

If you need to include header files from some other Python extension, you can take advantage of the fact that header files are installed in a consistent way by the Distutils `install_headers` command. For example, the Numerical Python header files are installed (on a standard Unix installation) to `/usr/local/include/python1.5/Numerical`. (The exact location will differ according to your platform and Python installation.) Since the Python include directory — `/usr/local/include/python1.5` in this case—is always
included in the search path when building Python extensions, the best approach is to write C code like

```c
#include <Numerical/arrayobject.h>
```

If you must put the Numerical include directory right into your header search path, though, you can find that directory using the Distutils `distutils.sysconfig` module:

```python
from distutils.sysconfig import get_python_inc
incdir = os.path.join(get_python_inc(plat_specific=1),
setup(...,
    Extension(..., include_dirs=[incdir]),
)
```

Even though this is quite portable—it will work on any Python installation, regardless of platform—it’s probably easier to just write your C code in the sensible way.

You can define and undefine pre-processor macros with the `define_macros` and `undef_macros` options. `define_macros` takes a list of (name, value) tuples, where name is the name of the macro to define (a string) and value is its value: either a string or None. (Defining a macro FOO to None is the equivalent of a bare `#define FOO` in your C source: with most compilers, this sets FOO to the string 1.) `undef_macros` is just a list of macros to undefine.

For example:

```python
Extension(...,
    define_macros=[('NDEBUG', '1'),
                  ('HAVE_STRFTIME', None)],
    undef_macros=['HAVE_FOO', 'HAVE_BAR'])
```
is the equivalent of having this at the top of every C source file:

```c
#define NDEBUG 1
#define HAVE_STRFTIME
#undef HAVE_FOO
#undef HAVE_BAR
```

### 2.3.4. Library options

You can also specify the libraries to link against when building your extension, and the directories to search for those libraries. The `libraries` option is a list of libraries to link against, `library_dirs` is a list of directories to search for libraries at link-time, and `runtime_library_dirs` is a list of directories to search for shared (dynamically loaded) libraries at run-time.

For example, if you need to link against libraries known to be in the standard library search path on target systems

```python
Extension(...,
    libraries=['gdbm', 'readline'])
```

If you need to link with libraries in a non-standard location, you'll have to include the location in `library_dirs`:

```python
Extension(...,
    library_dirs=['/usr/X11R6/lib'],
    libraries=['X11', 'Xt'])
```

(Again, this sort of non-portable construct should be avoided if you intend to distribute your code.)

### 2.3.5. Other options
There are still some other options which can be used to handle special cases.

The `optional` option is a boolean; if it is true, a build failure in the extension will not abort the build process, but instead simply not install the failing extension.

The `extra_objects` option is a list of object files to be passed to the linker. These files must not have extensions, as the default extension for the compiler is used.

`extra_compile_args` and `extra_link_args` can be used to specify additional command line options for the respective compiler and linker command lines.

`export_symbols` is only useful on Windows. It can contain a list of symbols (functions or variables) to be exported. This option is not needed when building compiled extensions: Distutils will automatically add `initmodule` to the list of exported symbols.

The `depends` option is a list of files that the extension depends on (for example header files). The build command will call the compiler on the sources to rebuild extension if any on this files has been modified since the previous build.
2.4. Relationships between Distributions and Packages

A distribution may relate to packages in three specific ways:

1. It can require packages or modules.
2. It can provide packages or modules.
3. It can obsolete packages or modules.

These relationships can be specified using keyword arguments to the `distutils.core.setup()` function.

Dependencies on other Python modules and packages can be specified by supplying the `requires` keyword argument to `setup()`. The value must be a list of strings. Each string specifies a package that is required, and optionally what versions are sufficient.

To specify that any version of a module or package is required, the string should consist entirely of the module or package name. Examples include 'mymodule' and 'xml.parsers.expat'.

If specific versions are required, a sequence of qualifiers can be supplied in parentheses. Each qualifier may consist of a comparison operator and a version number. The accepted comparison operators are:

```
<  >  ==
<=  >=  !=
```

These can be combined by using multiple qualifiers separated by commas (and optional whitespace). In this case, all of the qualifiers must be matched; a logical AND is used to combine the evaluations.
Let's look at a bunch of examples:

<table>
<thead>
<tr>
<th>Requires Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 1.0$</td>
<td>Only version 1.0 is compatible</td>
</tr>
<tr>
<td>$&gt;1.0, \neq 1.5.1, &lt;2.0$</td>
<td>Any version after 1.0 and before 2.0 is compatible, except 1.5.1</td>
</tr>
</tbody>
</table>

Now that we can specify dependencies, we also need to be able to specify what we provide that other distributions can require. This is done using the `provides` keyword argument to `setup()`. The value for this keyword is a list of strings, each of which names a Python module or package, and optionally identifies the version. If the version is not specified, it is assumed to match that of the distribution.

Some examples:

<table>
<thead>
<tr>
<th>Provides Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mypkg</td>
<td>Provide mypkg, using the distribution version</td>
</tr>
<tr>
<td>mypkg (1.1)</td>
<td>Provide mypkg version 1.1, regardless of the distribution version</td>
</tr>
</tbody>
</table>

A package can declare that it obsoletes other packages using the `obsoletes` keyword argument. The value for this is similar to that of the `requires` keyword: a list of strings giving module or package specifiers. Each specifier consists of a module or package name optionally followed by one or more version qualifiers. Version qualifiers are given in parentheses after the module or package name.

The versions identified by the qualifiers are those that are obsoleted by the distribution being described. If no qualifiers are given, all
versions of the named module or package are understood to be obsoleted.
2.5. Installing Scripts

So far we have been dealing with pure and non-pure Python modules, which are usually not run by themselves but imported by scripts.

Scripts are files containing Python source code, intended to be started from the command line. Scripts don’t require Distutils to do anything very complicated. The only clever feature is that if the first line of the script starts with `#!/` and contains the word “python”, the Distutils will adjust the first line to refer to the current interpreter location. By default, it is replaced with the current interpreter location. The `--executable` (or `-e`) option will allow the interpreter path to be explicitly overridden.

The `scripts` option simply is a list of files to be handled in this way. From the PyXML setup script:

```python
setup(...,
    scripts=['scripts/xmlproc_parse', 'scripts/xml...]
```

*Changed in version 3.1:* All the scripts will also be added to the MANIFEST file if no template is provided. See [Specifying the files to distribute](#).
2.6. Installing Package Data

Often, additional files need to be installed into a package. These files are often data that’s closely related to the package’s implementation, or text files containing documentation that might be of interest to programmers using the package. These files are called *package data*.

Package data can be added to packages using the `package_data` keyword argument to the `setup()` function. The value must be a mapping from package name to a list of relative path names that should be copied into the package. The paths are interpreted as relative to the directory containing the package (information from the `package_dir` mapping is used if appropriate); that is, the files are expected to be part of the package in the source directories. They may contain glob patterns as well.

The path names may contain directory portions; any necessary directories will be created in the installation.

For example, if a package should contain a subdirectory with several data files, the files can be arranged like this in the source tree:

```
setup.py
src/
    mypkg/
        __init__.py
        module.py
        data/
            tables.dat
            spoons.dat
            forks.dat
```

The corresponding call to `setup()` might be:
setup(...,
    packages=['mypkg'],
    package_dir={'mypkg': 'src/mypkg'},
    package_data={'mypkg': ['data/*.dat']},
)

*Changed in version 3.1:* All the files that match `package_data` will be added to the `MANIFEST` file if no template is provided. See `Specifying the files to distribute`. 
2.7. Installing Additional Files

The `data_files` option can be used to specify additional files needed by the module distribution: configuration files, message catalogs, data files, anything which doesn’t fit in the previous categories.

`data_files` specifies a sequence of `(directory, files)` pairs in the following way:

```python
setup(...,
    data_files=[
        ('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
        ('config', ['cfg/data.cfg'])
    ],
)
```

Each `(directory, files)` pair in the sequence specifies the installation directory and the files to install there.

Each file name in `files` is interpreted relative to the `setup.py` script at the top of the package source distribution. Note that you can specify the directory where the data files will be installed, but you cannot rename the data files themselves.

The `directory` should be a relative path. It is interpreted relative to the installation prefix (Python’s `sys.prefix` for system installations; `site.USER_BASE` for user installations). Distutils allows `directory` to be an absolute installation path, but this is discouraged since it is incompatible with the wheel packaging format. No directory information from `files` is used to determine the final location of the installed file; only the name of the file is used.

You can specify the `data_files` options as a simple sequence of
files without specifying a target directory, but this is not recommended, and the `install` command will print a warning in this case. To install data files directly in the target directory, an empty string should be given as the directory.

*Changed in version 3.1:* All the files that match `data_files` will be added to the `MANIFEST` file if no template is provided. See [Specifying the files to distribute](#).
2.8. Additional meta-data

The setup script may include additional meta-data beyond the name and version. This information includes:

<table>
<thead>
<tr>
<th>Meta-Data</th>
<th>Description</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name of the package</td>
<td>short string</td>
<td>(1)</td>
</tr>
<tr>
<td>version</td>
<td>version of this release</td>
<td>short string</td>
<td>(1)(2)</td>
</tr>
<tr>
<td>author</td>
<td>package author’s name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>author_email</td>
<td>email address of the package author</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer</td>
<td>package maintainer’s name</td>
<td>short string</td>
<td>(3)</td>
</tr>
<tr>
<td>maintainer_email</td>
<td>email address of the package maintainer</td>
<td>email address</td>
<td>(3)</td>
</tr>
<tr>
<td>url</td>
<td>home page for the package</td>
<td>URL</td>
<td>(1)</td>
</tr>
<tr>
<td>description</td>
<td>short, summary description of the package</td>
<td>short string</td>
<td></td>
</tr>
<tr>
<td>long_description</td>
<td>longer description of the package</td>
<td>long string</td>
<td>(4)</td>
</tr>
<tr>
<td>download_url</td>
<td>location where the package may be downloaded</td>
<td>URL</td>
<td></td>
</tr>
<tr>
<td>classifiers</td>
<td>a list of classifiers</td>
<td>list of</td>
<td>(6)(7)</td>
</tr>
<tr>
<td><strong>platforms</strong></td>
<td>a list of platforms</td>
<td>strings</td>
<td>(6)(8)</td>
</tr>
<tr>
<td><strong>keywords</strong></td>
<td>a list of keywords</td>
<td>list of strings</td>
<td>(6)(8)</td>
</tr>
<tr>
<td><strong>license</strong></td>
<td>license for the package</td>
<td>short string</td>
<td>(5)</td>
</tr>
</tbody>
</table>

Notes:

1. These fields are required.
2. It is recommended that versions take the form `major.minor[.patch[.sub]]`.
3. Either the author or the maintainer must be identified. If maintainer is provided, distutils lists it as the author in `PKG-INFO`.
4. The `long_description` field is used by PyPI when you are registering a package, to build its home page.
5. The `license` field is a text indicating the license covering the package where the license is not a selection from the “License” Trove classifiers. See the `Classifier` field. Notice that there's a `licence` distribution option which is deprecated but still acts as an alias for `license`.
6. This field must be a list.
7. The valid classifiers are listed on PyPI.
8. To preserve backward compatibility, this field also accepts a string. If you pass a comma-separated string 'foo, bar', it will be converted to `['foo', 'bar']`, Otherwise, it will be converted to a list of one string.

‘short string’
A single line of text, not more than 200 characters.

‘long string’
Multiple lines of plain text in reStructuredText format (see http://docutils.sourceforge.net/).

'list of strings'

See below.

Encoding the version information is an art in itself. Python packages generally adhere to the version format `major.minor[.patch][sub]`. The major number is 0 for initial, experimental releases of software. It is incremented for releases that represent major milestones in a package. The minor number is incremented when important new features are added to the package. The patch number increments when bug-fix releases are made. Additional trailing version information is sometimes used to indicate sub-releases. These are “a1,a2,…,aN” (for alpha releases, where functionality and API may change), “b1,b2,…,bN” (for beta releases, which only fix bugs) and “pr1,pr2,…,prN” (for final pre-release release testing). Some examples:

0.1.0

the first, experimental release of a package

1.0.1a2

the second alpha release of the first patch version of 1.0

classifiers must be specified in a list:

```python
setup(...,
    classifiers=[
        'Development Status :: 4 - Beta',
        'Environment :: Console',
        'Environment :: Web Environment',
        'Intended Audience :: End Users/Desktop',
        'Intended Audience :: Developers',
        'Intended Audience :: System Administrator',
        'License :: OSI Approved :: Python Software Foundation License',
        'Operating System :: MacOS :: MacOS X',
    ],
)
```
'Operating System :: Microsoft :: Windows',
'Operating System :: POSIX',
'Programming Language :: Python',
'Topic :: Communications :: Email',
'Topic :: Office/Business',
'Topic :: Software Development :: Bug Tracking',
]
)

Changed in version 3.7: `setup` now warns when `classifiers`, `keywords` or `platforms` fields are not specified as a list or a string.
2.9. Debugging the setup script

Sometimes things go wrong, and the setup script doesn’t do what the developer wants.

Distutils catches any exceptions when running the setup script, and print a simple error message before the script is terminated. The motivation for this behaviour is to not confuse administrators who don’t know much about Python and are trying to install a package. If they get a big long traceback from deep inside the guts of Distutils, they may think the package or the Python installation is broken because they don’t read all the way down to the bottom and see that it’s a permission problem.

On the other hand, this doesn’t help the developer to find the cause of the failure. For this purpose, the DISTUTILS_DEBUG environment variable can be set to anything except an empty string, and distutils will now print detailed information about what it is doing, dump the full traceback when an exception occurs, and print the whole command line when an external program (like a C compiler) fails.
1. An Introduction to Distutils

This document covers using the Distutils to distribute your Python modules, concentrating on the role of developer/distributor: if you're looking for information on installing Python modules, you should refer to the Installing Python Modules (Legacy version) chapter.
1.1. Concepts & Terminology

Using the Distutils is quite simple, both for module developers and for users/administrators installing third-party modules. As a developer, your responsibilities (apart from writing solid, well-documented and well-tested code, of course!) are:

- write a setup script (set up.py by convention)
- (optional) write a setup configuration file
- create a source distribution
- (optional) create one or more built (binary) distributions

Each of these tasks is covered in this document.

Not all module developers have access to a multitude of platforms, so it's not always feasible to expect them to create a multitude of built distributions. It is hoped that a class of intermediaries, called packagers, will arise to address this need. Packagers will take source distributions released by module developers, build them on one or more platforms, and release the resulting built distributions. Thus, users on the most popular platforms will be able to install most popular Python module distributions in the most natural way for their platform, without having to run a single setup script or compile a line of code.
1.2. A Simple Example

The setup script is usually quite simple, although since it's written in Python, there are no arbitrary limits to what you can do with it, though you should be careful about putting arbitrarily expensive operations in your setup script. Unlike, say, Autoconf-style configure scripts, the setup script may be run multiple times in the course of building and installing your module distribution.

If all you want to do is distribute a module called `foo`, contained in a file `foo.py`, then your setup script can be as simple as this:

```python
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
)
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the `setup()` function
- those keyword arguments fall into two categories: package metadata (name, version number) and information about what's in the package (a list of pure Python modules, in this case)
- modules are specified by module name, not filename (the same will hold true for packages and extensions)
- it's recommended that you supply a little more metadata, in particular your name, email address and a URL for the project (see section Writing the Setup Script for an example)

To create a source distribution for this module, you would create a setup script, `setup.py`, containing the above code, and run this
command from a terminal:

```python
python setup.py sdist
```

For Windows, open a command prompt window (Start ➔ Accessories) and change the command to:

```bash
setup.py sdist
```

**sdist** will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script `setup.py`, and your module `foo.py`. The archive file will be named `foo-1.0.tar.gz` (or `.zip`), and will unpack into a directory `foo-1.0`.

If an end-user wishes to install your `foo` module, all they have to do is download `foo-1.0.tar.gz` (or `.zip`), unpack it, and—from the `foo-1.0` directory—run

```bash
python setup.py install
```

which will ultimately copy `foo.py` to the appropriate directory for third-party modules in their Python installation.

This simple example demonstrates some fundamental concepts of the Distutils. First, both developers and installers have the same basic user interface, i.e. the setup script. The difference is which Distutils **commands** they use: the **sdist** command is almost exclusively for module developers, while **install** is more often for installers (although most developers will want to install their own code occasionally).

If you want to make things really easy for your users, you can create one or more built distributions for them. For instance, if you are running on a Windows machine, and want to make things easy for
other Windows users, you can create an executable installer (the most appropriate type of built distribution for this platform) with the **bdist_wininst** command. For example:

```bash
python setup.py bdist_wininst
```

will create an executable installer, `foo-1.0.win32.exe`, in the current directory.

Other useful built distribution formats are RPM, implemented by the **bdist_rpm** command, Solaris **pkgtool** (**bdist_pkgtool**), and HP-UX **swinstall** (**bdist_sdux**). For example, the following command will create an RPM file called `foo-1.0.noarch.rpm`:

```bash
python setup.py bdist_rpm
```

(The **bdist_rpm** command uses the **rpm** executable, therefore this has to be run on an RPM-based system such as Red Hat Linux, SuSE Linux, or Mandrake Linux.)

You can find out what distribution formats are available at any time by running

```bash
python setup.py bdist --help-formats
```
1.3. General Python terminology

If you’re reading this document, you probably have a good idea of what modules, extensions, and so forth are. Nevertheless, just to be sure that everyone is operating from a common starting point, we offer the following glossary of common Python terms:

module
the basic unit of code reusability in Python: a block of code imported by some other code. Three types of modules concern us here: pure Python modules, extension modules, and packages.

pure Python module
a module written in Python and contained in a single .py file (and possibly associated .pyc files). Sometimes referred to as a “pure module.”

extension module
a module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions. (Note that currently, the Distutils only handles C/C++ extensions for Python.)

package
a module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file __init__.py.

root package
the root of the hierarchy of packages. (This isn't really a package, since it doesn't have an __init__.py file. But we have to call it something.) The vast majority of the standard library is in the root
package, as are many small, standalone third-party modules that don't belong to a larger module collection. Unlike regular packages, modules in the root package can be found in many directories: in fact, every directory listed in sys.path contributes modules to the root package.
1.4. Distutils-specific terminology

The following terms apply more specifically to the domain of distributing Python modules using the Distutils:

module distribution
   a collection of Python modules distributed together as a single downloadable resource and meant to be installed *en masse*. Examples of some well-known module distributions are NumPy, SciPy, Pillow, or mxBase. (This would be called a *package*, except that term is already taken in the Python context: a single module distribution may contain zero, one, or many Python packages.)

pure module distribution
   a module distribution that contains only pure Python modules and packages. Sometimes referred to as a “pure distribution.”

non-pure module distribution
   a module distribution that contains at least one extension module. Sometimes referred to as a “non-pure distribution.”

distribution root
   the top-level directory of your source tree (or source distribution); the directory where `setup.py` exists. Generally `setup.py` will be run from this directory.
5. Creating Built Distributions

A “built distribution” is what you’re probably used to thinking of either as a “binary package” or an “installer” (depending on your background). It’s not necessarily binary, though, because it might contain only Python source code and/or byte-code; and we don’t call it a package, because that word is already spoken for in Python. (And “installer” is a term specific to the world of mainstream desktop systems.)

A built distribution is how you make life as easy as possible for installers of your module distribution: for users of RPM-based Linux systems, it’s a binary RPM; for Windows users, it’s an executable installer; for Debian-based Linux users, it’s a Debian package; and so forth. Obviously, no one person will be able to create built distributions for every platform under the sun, so the Distutils are designed to enable module developers to concentrate on their specialty—writing code and creating source distributions—while an intermediary species called packagers springs up to turn source distributions into built distributions for as many platforms as there are packagers.

Of course, the module developer could be their own packager; or the packager could be a volunteer “out there” somewhere who has access to a platform which the original developer does not; or it could be software periodically grabbing new source distributions and turning them into built distributions for as many platforms as the software has access to. Regardless of who they are, a packager uses the setup script and the bdist command family to generate built distributions.

As a simple example, if I run the following command in the Distutils source tree:
then the Distutils builds my module distribution (the Distutils itself in this case), does a “fake” installation (also in the build directory), and creates the default type of built distribution for my platform. The default format for built distributions is a “dumb” tar file on Unix, and a simple executable installer on Windows. (That tar file is considered “dumb” because it has to be unpacked in a specific location to work.)

Thus, the above command on a Unix system creates Distutils-1.0.plat.tar.gz; unpacking this tarball from the right place installs the Distutils just as though you had downloaded the source distribution and run python setup.py install. (The “right place” is either the root of the filesystem or Python’s prefix directory, depending on the options given to the bdist_dumb command; the default is to make dumb distributions relative to prefix.)

Obviously, for pure Python distributions, this isn’t any simpler than just running python setup.py install—but for non-pure distributions, which include extensions that would need to be compiled, it can mean the difference between someone being able to use your extensions or not. And creating “smart” built distributions, such as an RPM package or an executable installer for Windows, is far more convenient for users even if your distribution doesn’t include any extensions.

The bdist command has a --formats option, similar to the sdist command, which you can use to select the types of built distribution to generate: for example,

```
python setup.py bdist --format=zip
```
would, when run on a Unix system, create Distutils-1.0.<plat>.zip—again, this archive would be unpacked from the root directory to install the Distutils.

The available formats for built distributions are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gztar</td>
<td>gzipped tar file (.tar.gz)</td>
<td>(1)</td>
</tr>
<tr>
<td>bztar</td>
<td>bzipped tar file (.tar.bz2)</td>
<td></td>
</tr>
<tr>
<td>xztar</td>
<td>xzipped tar file (.tar.xz)</td>
<td></td>
</tr>
<tr>
<td>ztar</td>
<td>compressed tar file (.tar.Z)</td>
<td>(3)</td>
</tr>
<tr>
<td>tar</td>
<td>tar file (.tar)</td>
<td></td>
</tr>
<tr>
<td>zip</td>
<td>zip file (.zip)</td>
<td>(2),(4)</td>
</tr>
<tr>
<td>rpm</td>
<td>RPM</td>
<td>(5)</td>
</tr>
<tr>
<td>pkgtool</td>
<td>Solaris pkgtool</td>
<td></td>
</tr>
<tr>
<td>sdux</td>
<td>HP-UX swinstall</td>
<td></td>
</tr>
<tr>
<td>wininst</td>
<td>self-extracting ZIP file for Windows</td>
<td>(4)</td>
</tr>
<tr>
<td>msi</td>
<td>Microsoft Installer.</td>
<td></td>
</tr>
</tbody>
</table>

*Changed in version 3.5:* Added support for the xztar format.

Notes:

1. default on Unix
2. default on Windows
3. requires external **compress** utility.
4. requires either external **zip** utility or **zipfile** module (part of the standard Python library since Python 1.6)
5. requires external **rpm** utility, version 3.0.4 or better (use `rpm --`
version to find out which version you have)

You don't have to use the **bdist** command with the **--formats** option; you can also use the command that directly implements the format you're interested in. Some of these **bdist** “sub-commands” actually generate several similar formats; for instance, the **bdist_dumb** command generates all the “dumb” archive formats (tar, gztar, bztar, xztar, ztar, and zip), and **bdist_rpm** generates both binary and source RPMs. The **bdist** sub-commands, and the formats generated by each, are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bdist_dumb</strong></td>
<td>tar, gztar, bztar, xztar, ztar, zip</td>
</tr>
<tr>
<td><strong>bdist_rpm</strong></td>
<td>rpm, srpm</td>
</tr>
<tr>
<td><strong>bdist_wininst</strong></td>
<td>wininst</td>
</tr>
<tr>
<td><strong>bdist_msi</strong></td>
<td>msi</td>
</tr>
</tbody>
</table>

The following sections give details on the individual **bdist_** commands.
5.1. Creating RPM packages

The RPM format is used by many popular Linux distributions, including Red Hat, SuSE, and Mandrake. If one of these (or any of the other RPM-based Linux distributions) is your usual environment, creating RPM packages for other users of that same distribution is trivial. Depending on the complexity of your module distribution and differences between Linux distributions, you may also be able to create RPMs that work on different RPM-based distributions.

The usual way to create an RPM of your module distribution is to run the `bdist_rpm` command:

```
python setup.py bdist_rpm
```

or the `bdist` command with the `--format` option:

```
python setup.py bdist --format=rpm
```

The former allows you to specify RPM-specific options; the latter allows you to easily specify multiple formats in one run. If you need to do both, you can explicitly specify multiple `bdist_*` commands and their options:

```
python setup.py bdist_rpm --packager="John Doe <jdoe@example.org>
bdist_wininst --target-version="2.0"
```

Creating RPM packages is driven by a `.spec` file, much as using the Distutils is driven by the setup script. To make your life easier, the `bdist_rpm` command normally creates a `.spec` file based on the information you supply in the setup script, on the command line, and in
any Distutils configuration files. Various options and sections in the .spec file are derived from options in the setup script as follows:

<table>
<thead>
<tr>
<th>RPM .spec file option or section</th>
<th>Distutils setup script option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name</td>
</tr>
<tr>
<td>Summary (in preamble)</td>
<td>description</td>
</tr>
<tr>
<td>Version</td>
<td>version</td>
</tr>
<tr>
<td>Vendor</td>
<td>author and author_email, or — &amp; maintainer and maintainer_email</td>
</tr>
<tr>
<td>Copyright</td>
<td>license</td>
</tr>
<tr>
<td>Url</td>
<td>url</td>
</tr>
<tr>
<td>%description (section)</td>
<td>long_description</td>
</tr>
</tbody>
</table>

Additionally, there are many options in .spec files that don’t have corresponding options in the setup script. Most of these are handled through options to the bdist_rpm command as follows:

<table>
<thead>
<tr>
<th>RPM .spec file option or section</th>
<th>bdist_rpm option</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>release</td>
<td>“1”</td>
</tr>
<tr>
<td>Group</td>
<td>group</td>
<td>“Development/Libraries”</td>
</tr>
<tr>
<td>Vendor</td>
<td>vendor</td>
<td>(see above)</td>
</tr>
<tr>
<td>Packager</td>
<td>packager</td>
<td>(none)</td>
</tr>
<tr>
<td>Provides</td>
<td>provides</td>
<td>(none)</td>
</tr>
<tr>
<td>Requires</td>
<td>requires</td>
<td>(none)</td>
</tr>
</tbody>
</table>
Obviously, supplying even a few of these options on the command-line would be tedious and error-prone, so it’s usually best to put them in the setup configuration file, setup.cfg—see section Writing the Setup Configuration File. If you distribute or package many Python module distributions, you might want to put options that apply to all of them in your personal Distutils configuration file (~/.pydistutils.cfg). If you want to temporarily disable this file, you can pass the --no-user-cfg option to setup.py.

There are three steps to building a binary RPM package, all of which are handled automatically by the Distutils:

1. create a .spec file, which describes the package (analogous to the Distutils setup script; in fact, much of the information in the setup script winds up in the .spec file)
2. create the source RPM
3. create the “binary” RPM (which may or may not contain binary code, depending on whether your module distribution contains Python extensions)

Normally, RPM bundles the last two steps together; when you use the Distutils, all three steps are typically bundled together.

If you wish, you can separate these three steps. You can use the --spec-only option to make bdist_rpm just create the .spec file and exit; in this case, the .spec file will be written to the “distribution
directory”—normally dist/,” but customizable with the --dist-dir option. (Normally, the .spec file winds up deep in the “build tree,” in a temporary directory created by bdist_rpm.)
5.2. Creating Windows Installers

Executable installers are the natural format for binary distributions on Windows. They display a nice graphical user interface, display some information about the module distribution to be installed taken from the metadata in the setup script, let the user select a few options, and start or cancel the installation.

Since the metadata is taken from the setup script, creating Windows installers is usually as easy as running:

```
python setup.py bdist_wininst
```

or the `bdist` command with the `--formats` option:

```
python setup.py bdist --formats=wininst
```

If you have a pure module distribution (only containing pure Python modules and packages), the resulting installer will be version independent and have a name like `foo-1.0.win32.exe`. These installers can even be created on Unix platforms or Mac OS X.

If you have a non-pure distribution, the extensions can only be created on a Windows platform, and will be Python version dependent. The installer filename will reflect this and now has the form `foo-1.0.win32-py2.0.exe`. You have to create a separate installer for every Python version you want to support.

The installer will try to compile pure modules into bytecode after installation on the target system in normal and optimizing mode. If you don't want this to happen for some reason, you can run the `bdist_wininst` command with the `--no-target-compile` and/or
the --no-target-optimize option.

By default the installer will display the cool “Python Powered” logo when it is run, but you can also supply your own 152x261 bitmap which must be a Windows .bmp file with the --bitmap option.

The installer will also display a large title on the desktop background window when it is run, which is constructed from the name of your distribution and the version number. This can be changed to another text by using the --title option.

The installer file will be written to the “distribution directory” — normally dist/, but customizable with the --dist-dir option.
5.3. Cross-compiling on Windows

Starting with Python 2.6, distutils is capable of cross-compiling between Windows platforms. In practice, this means that with the correct tools installed, you can use a 32bit version of Windows to create 64bit extensions and vice-versa.

To build for an alternate platform, specify the --plat-name option to the build command. Valid values are currently ‘win32’, and ‘win-amd64’. For example, on a 32bit version of Windows, you could execute:

```
python setup.py build --plat-name=win-amd64
```

to build a 64bit version of your extension. The Windows Installers also support this option, so the command:

```
python setup.py build --plat-name=win-amd64 bdist_wininst
```

would create a 64bit installation executable on your 32bit version of Windows.

To cross-compile, you must download the Python source code and cross-compile Python itself for the platform you are targeting - it is not possible from a binary installation of Python (as the .lib etc file for other platforms are not included.) In practice, this means the user of a 32 bit operating system will need to use Visual Studio 2008 to open the PCbuild/PCbuild.sln solution in the Python source tree and build the “x64” configuration of the ‘pythoncore’ project before cross-compiling extensions is possible.

Note that by default, Visual Studio 2008 does not install 64bit
compilers or tools. You may need to reexecute the Visual Studio setup process and select these tools (using Control Panel->[Add/Remove] Programs is a convenient way to check or modify your existing install.)

5.3.1. The Postinstallation script

Starting with Python 2.3, a postinstallation script can be specified with the --install-script option. The basename of the script must be specified, and the script filename must also be listed in the scripts argument to the setup function.

This script will be run at installation time on the target system after all the files have been copied, with `argv[1]` set to `-install`, and again at uninstallation time before the files are removed with `argv[1]` set to `-remove`.

The installation script runs embedded in the windows installer, every output (`sys.stdout`, `sys.stderr`) is redirected into a buffer and will be displayed in the GUI after the script has finished.

Some functions especially useful in this context are available as additional built-in functions in the installation script.

**directory_created**(path)
**file_created**(path)

These functions should be called when a directory or file is created by the postinstall script at installation time. It will register `path` with the uninstaller, so that it will be removed when the distribution is uninstalled. To be safe, directories are only removed if they are empty.

**get_special_folder_path**(cIDL_string)
This function can be used to retrieve special folder locations on Windows like the Start Menu or the Desktop. It returns the full path to the folder. *csidl_string* must be one of the following strings:

```
"CSIDL_APPDATA"
"CSIDL_COMMON_STARTMENU"
"CSIDL_STARTMENU"
"CSIDL_COMMON_DESKTOPDIRECTORY"
"CSIDL_DESKTOPDIRECTORY"
"CSIDL_COMMON_STARTUP"
"CSIDL_STARTUP"
"CSIDL_COMMON_PROGRAMS"
"CSIDL_PROGRAMS"
"CSIDL_FONTS"
```

If the folder cannot be retrieved, `OSError` is raised.

Which folders are available depends on the exact Windows version, and probably also the configuration. For details refer to Microsoft's documentation of the `SHGetSpecialFolderPath()` function.

```python
create_shortcut(target, description, filename[, arguments[, workdir[, iconpath[, iconindex]]]])
```

This function creates a shortcut. `target` is the path to the program to be started by the shortcut. `description` is the description of the shortcut. `filename` is the title of the shortcut that the user will see. `arguments` specifies the command line arguments, if any. `workdir` is the working directory for the program. `iconpath` is the file containing
the icon for the shortcut, and \textit{iconindex} is the index of the icon in the file \textit{iconpath}. Again, for details consult the Microsoft documentation for the IShellLink interface.
5.4. Vista User Access Control (UAC)

Starting with Python 2.6, bdist_wininst supports a --user-access-control option. The default is ‘none’ (meaning no UAC handling is done), and other valid values are ‘auto’ (meaning prompt for UAC elevation if Python was installed for all users) and ‘force’ (meaning always prompt for elevation).
6. The Python Package Index (PyPI)

The Python Package Index (PyPI) stores meta-data describing distributions packaged with distutils, as well as package data like distribution files if a package author wishes.

Distutils provides the register and upload commands for pushing meta-data and distribution files to PyPI, respectively. See Distutils commands for information on these commands.
6.1. PyPI overview

PyPI lets you submit any number of versions of your distribution to the index. If you alter the meta-data for a particular version, you can submit it again and the index will be updated.

PyPI holds a record for each (name, version) combination submitted. The first user to submit information for a given name is designated the Owner of that name. Changes can be submitted through the register command or through the web interface. Owners can designate other users as Owners or Maintainers. Maintainers can edit the package information, but not designate new Owners or Maintainers.

By default PyPI displays only the newest version of a given package. The web interface lets one change this default behavior and manually select which versions to display and hide.

For each version, PyPI displays a home page. The home page is created from the long_description which can be submitted via the register command. See PyPI package display for more information.
6.2. Distutils commands

Distutils exposes two commands for submitting package data to PyPI: the register command for submitting meta-data to PyPI and the upload command for submitting distribution files. Both commands read configuration data from a special file called a .pypirc file.

6.2.1. The register command

The distutils command register is used to submit your distribution’s meta-data to an index server. It is invoked as follows:

```
python setup.py register
```

Distutils will respond with the following prompt:

```
running register
We need to know who you are, so please choose either
  1. use your existing login,
  2. register as a new user,
  3. have the server generate a new password for you
  4. quit
Your selection [default 1]:
```

Note: if your username and password are saved locally, you will not see this menu. Also, refer to The .pypirc file for how to store your credentials in a .pypirc file.

If you have not registered with PyPI, then you will need to do so now. You should choose option 2, and enter your details as required. Soon after submitting your details, you will receive an email which will be used to confirm your registration.
Once you are registered, you may choose option 1 from the menu. You will be prompted for your PyPI username and password, and `register` will then submit your meta-data to the index.

See Additional command options for options to the `register` command.

### 6.2.2. The upload command

The distutils command `upload` pushes the distribution files to PyPI.

The command is invoked immediately after building one or more distribution files. For example, the command

```bash
python setup.py sdist bdist_wininst upload
```

will cause the source distribution and the Windows installer to be uploaded to PyPI. Note that these will be uploaded even if they are built using an earlier invocation of `setup.py`, but that only distributions named on the command line for the invocation including the `upload` command are uploaded.

If a `register` command was previously called in the same command, and if the password was entered in the prompt, `upload` will reuse the entered password. This is useful if you do not want to store a password in clear text in a `.pypirc` file.

You can use the `--sign` option to tell `upload` to sign each uploaded file using GPG (GNU Privacy Guard). The `gpg` program must be available for execution on the system PATH. You can also specify which key to use for signing using the `--identity=name` option.

See Additional command options for additional options to the `upload` command.
6.2.3. Additional command options

This section describes options common to both the **register** and **upload** commands.

The **--repository** or **-r** option lets you specify a PyPI server different from the default. For example:

```bash
python setup.py sdist bdist_wheel upload -r https://...
```

For convenience, a name can be used in place of the URL when the .pypirc file is configured to do so. For example:

```bash
python setup.py register -r other
```

See [The .pypirc file](#) for more information on defining alternate servers.

The **--show-response** option displays the full response text from the PyPI server, which is useful when debugging problems with registering and uploading.

6.2.4. The .pypirc file

The **register** and **upload** commands both check for the existence of a .pypirc file at the location $HOME/.pypirc. If this file exists, the command uses the username, password, and repository URL configured in the file. The format of a .pypirc file is as follows:

```conf
[distutils]
index-servers = pypi
```
The *distutils* section defines an *index-servers* variable that lists the name of all sections describing a repository.

Each section describing a repository defines three variables:

- *repository*, that defines the url of the PyPI server. Defaults to `https://upload.pypi.org/legacy/`.
- *username*, which is the registered username on the PyPI server.
- *password*, that will be used to authenticate. If omitted the user will be prompt to type it when needed.

If you want to define another server a new section can be created and listed in the *index-servers* variable:

```ini
[distutils]
index-servers =
    pypi
    other

[pypi]
repository: <repository-url>
username: <username>
password: <password>

[other]
repository: https://example.com/pypi
username: <username>
password: <password>
```

This allows the `register` and `upload` commands to be called with the -
-repository option as described in Additional command options.

Specifically, you might want to add the PyPI Test Repository to your .pypirc to facilitate testing before doing your first upload to PyPI itself.
6.3. PyPI package display

The long_description field plays a special role at PyPI. It is used by the server to display a home page for the registered package.

If you use the reStructuredText syntax for this field, PyPI will parse it and display an HTML output for the package home page.

The long_description field can be attached to a text file located in the package:

```python
from distutils.core import setup

with open('README.txt') as file:
    long_description = file.read()

setup(name='Distutils',
      long_description=long_description)
```

In that case, README.txt is a regular reStructuredText text file located in the root of the package besides setup.py.

To prevent registering broken reStructuredText content, you can use the rst2html program that is provided by the docutils package and check the long_description from the command line:

```
$ python setup.py --long-description | rst2html.py > output.html
```

docutils will display a warning if there's something wrong with your syntax. Because PyPI applies additional checks (e.g. by passing --no-raw to rst2html.py in the command above), being able to run the command above without warnings does not guarantee that PyPI
will convert the content successfully.
7. Examples

This chapter provides a number of basic examples to help get started with distutils. Additional information about using distutils can be found in the Distutils Cookbook.

See also:

Distutils Cookbook
Collection of recipes showing how to achieve more control over distutils.
7.1. Pure Python distribution (by module)

If you’re just distributing a couple of modules, especially if they don’t live in a particular package, you can specify them individually using the `py_modules` option in the setup script.

In the simplest case, you’ll have two files to worry about: a setup script and the single module you’re distributing, `foo.py` in this example:

```
<root>/
    setup.py
    foo.py
```

(In all diagrams in this section, `<root>` will refer to the distribution root directory.) A minimal setup script to describe this situation would be:

```python
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
)
```

Note that the name of the distribution is specified independently with the `name` option, and there’s no rule that says it has to be the same as the name of the sole module in the distribution (although that’s probably a good convention to follow). However, the distribution name is used to generate filenames, so you should stick to letters, digits, underscores, and hyphens.

Since `py_modules` is a list, you can of course specify multiple modules, eg. if you’re distributing modules `foo` and `bar`, your setup might look like this:
and the setup script might be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      py_modules=['foo', 'bar'],
      )
```

You can put module source files into another directory, but if you have enough modules to do that, it's probably easier to specify modules by package rather than listing them individually.
7.2. Pure Python distribution (by package)

If you have more than a couple of modules to distribute, especially if they are in multiple packages, it's probably easier to specify whole packages rather than individual modules. This works even if your modules are not in a package; you can just tell the Distutils to process modules from the root package, and that works the same as any other package (except that you don't have to have an `__init__.py` file).

The setup script from the last example could also be written as

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=[''],
)
```

(The empty string stands for the root package.)

If those two files are moved into a subdirectory, but remain in the root package, e.g.:

```plaintext
<root>/
  setup.py
  src/    foo.py
          bar.py
```

then you would still specify the root package, but you have to tell the Distutils where source files in the root package live:

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'': 'src'},
```
More typically, though, you will want to distribute multiple modules in the same package (or in sub-packages). For example, if the foo and bar modules belong in package foobar, one way to layout your source tree is

```python
from distutils.core import setup
setup(name='foobar',
    version='1.0',
    packages=['foobar'],
)
```

If you want to put modules in directories not named for their package, then you need to use the `package_dir` option again. For example, if the `src` directory holds modules in the `foobar` package:

```python
from distutils.core import setup
setup(name='foobar',
    version='1.0',
    package_dir={'foobar': 'src'},
    packages=['foobar'],
)
```
an appropriate setup script would be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': 'src'},
      packages=['foobar'],
)
```

Or, you might put modules from your main package right in the distribution root:

```
<root>/
    setup.py
    __init__.py
    foo.py
    bar.py
```

in which case your setup script would be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': ''},
      packages=['foobar'],
)
```

(The empty string also stands for the current directory.)

If you have sub-packages, they must be explicitly listed in `packages`, but any entries in `package_dir` automatically extend to sub-packages. (In other words, the Distutils does not scan your source tree, trying to figure out which directories correspond to Python packages by looking for `__init__.py` files.) Thus, if the default layout grows a sub-package:
then the corresponding setup script would be

```python
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=['foobar', 'foobar.subfoo'],
      )
```
7.3. Single extension module

Extension modules are specified using the `ext_modules` option. `package_dir` has no effect on where extension source files are found; it only affects the source for pure Python modules. The simplest case, a single extension module in a single C source file, is:

```python
classification<root>/
    setup.py
    foo.c
```

If the `foo` extension belongs in the root package, the setup script for this could be

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
)
```

If the extension actually belongs in a package, say `foopkg`, then

With exactly the same source tree layout, this extension can be put in the `foopkg` package simply by changing the name of the extension:

```python
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foopkg.foo', ['foo.c'])]
)
```
7.4. Checking a package

The check command allows you to verify if your package meta-data meet the minimum requirements to build a distribution.

To run it, just call it using your `setup.py` script. If something is missing, `check` will display a warning.

Let's take an example with a simple script:

```python
from distutils.core import setup
setup(name='foobar')
```

Running the `check` command will display some warnings:

```
$ python setup.py check
running check
warning: check: missing required meta-data: version,
warning: check: missing meta-data: either (author and
    (maintainer and maintainer_email) must be s
```

If you use the reStructuredText syntax in the `long_description` field and `docutils` is installed you can check if the syntax is fine with the `check` command, using the `restructuredtext` option.

For example, if the `setup.py` script is changed like this:

```python
from distutils.core import setup
desc = """
My description
"""
```
This is the description of the ``foobar`` package. 

````
setup(name='foobar', version='1', author='tarek',
      author_email='tarek@ziade.org',
      url='http://example.com', long_description=desc)
````

Where the long description is broken, check will be able to detect it by using the docutils parser:

```
$ python setup.py check --restructuredtext
running check
warning: check: Title underline too short. (line 2)
warning: check: Could not finish the parsing.
```
7.5. Reading the metadata

The `distutils.core.setup()` function provides a command-line interface that allows you to query the metadata fields of a project through the `setup.py` script of a given project:

```bash
$ python setup.py --name distribute
```

This call reads the `name` metadata by running the `distutils.core.setup()` function. Although, when a source or binary distribution is created with Distutils, the metadata fields are written in a static file called `PKG-INFO`. When a Distutils-based project is installed in Python, the `PKG-INFO` file is copied alongside the modules and packages of the distribution under `NAME-VERSION-pyX.X.egg-info`, where `NAME` is the name of the project, `VERSION` its version as defined in the Metadata, and `pyX.X` the major and minor version of Python like 2.7 or 3.2.

You can read back this static file, by using the `distutils.dist.DistributionMetadata` class and its `read_pkg_file()` method:

```python
>>> from distutils.dist import DistributionMetadata
>>> metadata = DistributionMetadata()
>>> metadata.read_pkg_file(open('distribute-0.6.8-py2.7.egg-info'))
>>> metadata.name
'distribute'
>>> metadata.version
'0.6.8'
>>> metadata.description
'Easily download, build, install, upgrade, and uninstall Python packages'
```
Notice that the class can also be instantiated with a metadata file path to loads its values:

```python
>>> pkg_info_path = 'distribute-0.6.8-py2.7.egg-info'
>>> DistributionMetadata(pkg_info_path).name
'distribute'
```